国外计算机科学教材系列

类型和程序设计语言

Types and Programming Languages

Types and Programming Languages

Benjamin C. Pierce

[美] Benjamin C. Pierce 著
马世龙 眭跃飞 等译
眭跃飞 审校




電子工業出版社
Publishing House of Electronics Industry
http://www.phei.com.cn



类型和程序设计语言

Types and Programming Languages

类型系统是一个可行的语法工具，根据程序所计算值的种类将程序词语进行分类，从而自动检查出错误行为。类型系统的研究和从类型理论角度对程序设计语言的研究，在软件工程语言的设计、高性能编译器和安全性方面都有着重要的应用。

本书对计算机科学和程序设计语言基础理论中的类型系统进行了详细地介绍。所提到的方法都很实用，每个新的概念都以程序实例加以解释，偏重理论的方面根据实现的需要有所取舍。每章都附有大量的练习和相应的解答，并且可在Web上找到相应的实现程序。章节之间的关系有明确的指示，读者可以据此选择适合自己的内容进行阅读。

类型是计算机程序语言的酵母，若少了它，程序难以被计算机消化。这本优秀的图书集应用、理论和实现为一体，通过类型指引我们走进丰富的程序语言世界。本书的作者在应用、理论和实现方面有着丰富的经验。

——Robin Milner，剑桥大学计算机实验室

只有出色的学者才能写出如此严谨、清晰的书籍，才能将理论与实现技术融为一体，才能体现出丰富的教学经验，才能算得上该领域的专家。

——John Reynolds，卡内基 · 梅隆大学计算机学院

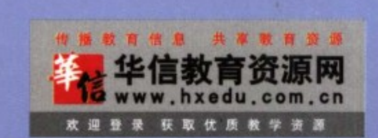
Pierce的书不但对程序语言中的类型进行了详细介绍，而且将理论和实践有机地结合起来，提供了一个成功的典范。未来许多年间，此书必将成为权威参考书目。

——Robert Harper，卡内基 · 梅隆大学计算机系教授

本书是经过了对多个问题仔细推敲、选择后写成的。它侧重于实用，同时也不缺乏必要的理论。书中的练习充分考虑了初级读者、高级读者、程序员和理论研究人员的不同需要。

——Henk Barendregt，荷兰纳米根大学数学与计算机科学教师









责任编辑：李秦华
责任美编：毛惠庚



ISBN 7-121-01149-2　定价：58.00元

国外计算机科学教材系列

类型和程序设计语言

Types and Programming Languages

[美] Benjamin C. Pierce 著

马世龙 眭跃飞 等译

眭跃飞 审校

電子工業出版社

Publishing House of Electronics Industry

北京 · BEIJING

内容简介

类型理论在程序设计语言的发展中起着举足轻重的作用，成熟的类型系统可以帮助完善程序设计本身，帮助运行系统检查程序中的语义错误。

要理解类型系统在程序设计语言中发挥的作用，本书将是首选读物。本书内容覆盖基本操作语义及其相关证明技巧、无类型 lambda 演算、简单类型系统、全称多态和存在多态、类型重构、子类型化、囿界量词、递归类型、类型算子等内容。本书既注重内容的广度，也注重内容的深度，实用性强。在引入语言的语法对象时先举例，然后给出形式定义及基本证明，在对理论的进一步研究后给出了类型检查算法，并对每种算法都给出了OCaml程序的具体实现。本书对类型理论中的概念都有详细的阐述，为读者提供了一个进一步理论学习的基础。本书内容广泛，读者可以根据自己的需要有选择地深入阅读。

本书适合从事程序设计的研究人员和开发人员，以及程序设计语言和类型理论的研究人员阅读。可作为计算机专业高年级学生、研究生的学习教材。

版权贸易合同登记号　图字：01-2002-6448

图书在版编目（CIP）数据

类型和程序设计语言 /（美）皮尔斯（Pierce, B. C.）著；马世龙，眭跃飞等译.
北京：电子工业出版社，2005.5
（国外计算机科学教材系列）
书名原文：Types and Programming Languages
ISBN 7-121-01149-2

Ⅰ.类…　Ⅱ.①皮…②马…③眭…　Ⅲ.类型学（语言学）- 应用 - 程序语言 - 教材　Ⅳ. TP312

中国版本图书馆 CIP 数据核字（2005）第 038577 号

责任编辑：李秦华
印　　刷：北京东光印刷厂
出版发行：电子工业出版社
　　　　　北京市海淀区万寿路 173 信箱　邮编：100036
经　　销：各地新华书店
开　　本：787 × 1092　1/16　　印张：27.75　字数：710 千字
印　　次：2005 年 5 月第 1 次印刷
定　　价：58.00 元

凡购买电子工业出版社的图书，如有缺损问题，请向购买书店调换；若书店售缺，请与本社发行部联系。联系电话：（010）68279077。质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

出 版 说 明

21世纪初的5至10年是我国国民经济和社会发展的重要时期，也是信息产业快速发展的关键时期。在我国加入WTO后的今天，培养一支适应国际化竞争的一流IT人才队伍是我国高等教育的重要任务之一。信息科学和技术方面人才的优劣与多寡，是我国面对国际竞争时成败的关键因素。

当前，正值我国高等教育特别是信息科学领域的教育调整、变革的重大时期，为使我国教育体制与国际化接轨，有条件的高等院校正在为某些信息学科和技术课程使用国外优秀教材和优秀原版教材，以使我国在计算机教学上尽快赶上国际先进水平。

电子工业出版社秉承多年来引进国外优秀图书的经验，翻译出版了“国外计算机科学教材系列”丛书，这套教材覆盖学科范围广、领域宽、层次多，既有本科专业课程教材，也有研究生课程教材，以适应不同院系、不同专业、不同层次的师生对教材的需求，广大师生可自由选择和自由组合使用。这些教材涉及的学科方向包括网络与通信、操作系统、计算机组织与结构、算法与数据结构、数据库与信息处理、编程语言、图形图像与多媒体、软件工程等。同时，我们也适当引进了一些优秀英文原版教材，本着翻译版本和英文原版并重的原则，对重点图书既提供英文原版又提供相应的翻译版本。

在图书选题上，我们大都选择国外著名出版公司出版的高校教材，如Pearson Education培生教育出版集团、麦格劳－希尔教育出版集团、麻省理工学院出版社、剑桥大学出版社等。撰写教材的许多作者都是蜚声世界的教授、学者，如道格拉斯·科默(Douglas E. Comer)、威廉·斯托林斯(William Stallings)、哈维・戴特尔（Harvey M. Deitel)、尤利斯・布莱克（Uyless Black）等。

为确保教材的选题质量和翻译质量，我们约请了清华大学、北京大学、北京航空航天大学、复旦大学、上海交通大学、南京大学、浙江大学、哈尔滨工业大学、华中科技大学、西安交通大学、国防科学技术大学、解放军理工大学等著名高校的教授和骨干教师参与了本系列教材的选题、翻译和审校工作。他们中既有讲授同类教材的骨干教师、博士，也有积累了几十年教学经验的老教授和博士生导师。

在该系列教材的选题、翻译和编辑加工过程中，为提高教材质量，我们做了大量细致的工作，包括对所选教材进行全面论证；选择编辑时力求达到专业对口；对排版、印制质量进行严格把关。对于英文教材中出现的错误，我们通过与作者联络和网上下载勘误表等方式，逐一进行了修订。

此外，我们还将与国外著名出版公司合作，提供一些教材的教学支持资料，希望能为授课老师提供帮助。今后，我们将继续加强与各高校教师的密切联系，为广大师生引进更多的国外优秀教材和参考书，为我国计算机科学教学体系与国际教学体系的接轨做出努力。

电子工业出版社

教材出版委员会

译 者 序

类型系统的研究，以及从类型论的角度研究程序语言已经是一个充满活力的研究领域，主要应用于软件工程、语言设计、高性能编译器的实现和安全。本书以一种很容易理解的方式介绍这一领域所涉及的基本定义、理论结果（从基本lambda演算、类型理论和程序语言理论）和实现技术。

现代软件工程广泛地使用形式化方法，确保系统按照隐式或显式的说明方式运行。这方面出现过一些过于理论化的框架，如Hoare逻辑、代数说明语言、模态逻辑和指称语义等。由于这些理论实用性不强，尤其难以被程序员所接受，所以出现了一些实用的技术，如将自动检查器放到编译器、连接器或程序分析器中的技术。至今最流行、最完善的轻量级形式化方法是类型系统，也正是作者在本书中讨论的主要内容。

作者将类型系统定义为“一种可行的语法工具，它可以根据计算值的种类对词语进行分类，从而证明某程序行为不会发生”。形式的定义可能会误导读者认为类型系统只是在理论上重要。事实上，类型系统已经成为程序语言设计的一个重要方面。类型系统为编译器提供了类型检查的手段，可以检测程序员犯下的由于类型错误而引起的语义错误，这样就可避免不规范行为的产生，如程序崩溃。我们在互联网上遇到的许多安全漏洞其实就是由于没有正确检测到类型错误而引起的。通常转而使用类型成熟语言的程序员开始时都会觉得十分惊讶，因为他们的程序无需额外的调试立即就能运行，可见类型系统有多么重要。

本书带领读者走进了一个关于类型系统设计、推理和实现的有趣领域。作者通过使用操作语义和OCaml语言举例（这些例子在书上提供的相关网站上都可以查到），将理论问题表达得清晰、易懂。

作者按照通常的方式介绍带有类型的程序语言中的语法对象，全书贯穿许多启发性例子、严格的定义、基本性质的证明、类型检查算法的描述、可靠性、完备性和可终止性的证明，并给出算法的具体实现。讨论的问题涵盖了lambda演算、简单类型、子类型、递归类型、多态和高阶类型。尽管强调的重点是函数式语言，如Haskell和ML的类型系统，但是也谈到了与理论发展有关的主流语言，如Java和C++语言之间的关系。本书理论与实践结合得非常紧密，对学习程序语言及培养语言设计人员来说帮助很大。从无类型系统到有类型系统，从简单类型到复杂类型，对各类语言特征都进行了详细地阐述，对命题和定理的证明也十分严谨。作者还开发了一套TinkerType系统，完成对书中所举的每个例子进行类型检查器的检查，并给出执行结果。该书如果有欠缺之处，那就是没有将流行的程序语言的特色、优点及缺点加以严格评论。

本书采用合适的排版式样和丰富的字体提高可读性，例如可以清晰地表达复杂冗长的lambda归约序列、操作语义和程序实例；同时给出了程序、规则和性质在表示上的差别。本书

包括不同层次难度的练习，并提供了详细的参考文献，以帮助高年级本科生、研究生以及相关学者学习和参考。初读本书的读者如果感到理解上有些困难，这并不奇怪。因为正确理解书中的内容需要读者具有一定的理论基础和程序设计的实践经验，融会贯通则更需要下一番功夫。

毫无疑问，本书是关于类型理论程序设计的重要著作，在此领域中的地位是无可替代的。

北京航空航天大学博士生导师马世龙教授及中科院计算所博士生导师眭跃飞教授组织博士研究生和硕士研究生对类型理论进行了长期的研究和讨论，在此过程中翻译了本书。其中主要翻译者有：前言及第1章到第14章由眭跃飞翻译，第15章到第21章由马丽和王婷翻译，第22章到第26章由马丽翻译，第27章到第30章由马骏翻译，第31章到第32章及附录由陈锬翻译。全书由马世龙教授统稿，眭跃飞教授审定。另外感谢毛俏琳在前期统稿中所做的工作。

由于译者水平有限，以及一些术语的翻译目前尚缺乏规范，错误之处望广大读者批评指正。

前　言

对类型系统的研究,以及从类型理论的角度研究程序语言已成为一种充满活力的领域,它们主要应用于软件工程、语言设计、高性能编译器的实现和安全。本书将对该领域的基本概念,研究成果及技术手段展开全面的阐述。

读者对象

本书针对两类读者:(1)从事程序语言和类型理论研究的研究生和研究人员;(2)希望了解程序语言理论中关键概念的计算机科学领域研究生和高年级本科生。对于前一类读者,本书覆盖了该领域的大部分内容,为读者深入研究提供了依据。对于后一类读者,提供了介绍性的材料和例子以及分析。本书既可以作为一般研究生水平的教材,也可以作为程序语言的高级研讨班的教材。

本书的目标

目标之一:在内容上力求覆盖一些核心概念,如基本操作语义及相关证明技巧、无类型 lambda 演算、简单类型系统、全称多态和存在多态、类型重构、子类型化、囿量词、递归类型、类型算子,以及许多其他问题的简短讨论。

目的之二:从语用的角度,注重程序语言中类型系统的使用,用大量的篇幅对一些可能包括在类型化 lambda 演算中的更精确问题(如指称语义)进行讨论。其中计算基础是值调用 lambda 演算,它与当前大部分语言相符,并且容易扩展为引用和异常等命令式结构。对每种语言的特征,关注它的实用价值、语言所含安全性的证明技巧,以及如何具体实现等方面的内容,尤其是类型检查算法的设计和分析。

目标之三:关注领域的多样性。本书覆盖了许多独立的问题和几个深刻理解的复合问题,但没有试图将所有的问题都溶入一个单一的系统中。只是对这些问题的子集提出了统一的表示方式,比如,虽然箭头类型变化多样,但在纯类型系统中均以统一的方式处理,由于整个领域发展太快,无法对其完全系统化。

本书在设计上充分考虑方便读者学习,无论是在课堂上还是以自学的方式。对大部分习题都给出了全面的解答。核心定义组织成一个图表以便于参考。概念和系统之间的依赖关系表现得尽可能清晰明了。在书的最后还提供了大量的参考书目。

最后,本书恪守诚实的原则。书中讨论的所有系统(除个别稍微提及的系统外)均有实现。每章都有从机器的角度检查所举例子的类型检查器和解释器。这些实现方式均可从本书的网站上获得,可用做编程练习,进行扩充实验以及作为更大的课堂报告内容。

为达到这些目标,其他的一些因素往往被忽略了。比如,最重要的一点就是内容的全面性。在一本书中想覆盖编程语言和类型系统的所有领域是不可能的,本书也不例外。本书只关注核心概念的详细讨论。还有,本书不考虑类型检查算法实际的效率,因为它并非一本关于如何实现具有工业力度的编译器或类型检查器的书籍。

本书的组织结构

本书的第一部分讨论无类型系统。在一个非常简单的仅含数字型和布尔型的语言中引入一些基本概念:抽象语法、归纳定义和证明、推论规则和操作语义,然后对无类型 lambda 演算重复引入这些概念进行讨论。第二部分考虑简单类型 lambda 演算和一些基本语言特征,如乘积、求和、记录、变式、引用和异常等。其中包括关于类型算术表达式一章引入了类型安全性的思想。还包括一章(可选学)用 Tait 的方法给出了简单类型 lambda 演算的规范化证明。第三部分讨论子类型的基本机制:包括一个元理论和两个扩展的实例研究。第四部分讨论递归类型,包含了简单同构递归和复杂相等递归形式。该部分的第 21 章讨论了在共归纳的数学框架中含相等递归和子类型的系统元理论。第五部分讨论多态性,包括 ML 形式的类型重构,功能更强大系统 F 和不可预言的多态性,存在量词及其与抽象数据类型的联系,带囿量词的系统中多态与子类型的组合,等等。第六部分讨论了类型算子。其中一章讨论基本概念;接下来的一章研究系统 F_{ω} 及其元理论;下一章将类型算子与囿量词结合起来提出系统 $F^{\omega}_{<:}$;最后一章为一个实例学习。

图 P.1 给出了章节之间的依赖关系。浅色箭头表示后一章部分依赖于前一章。

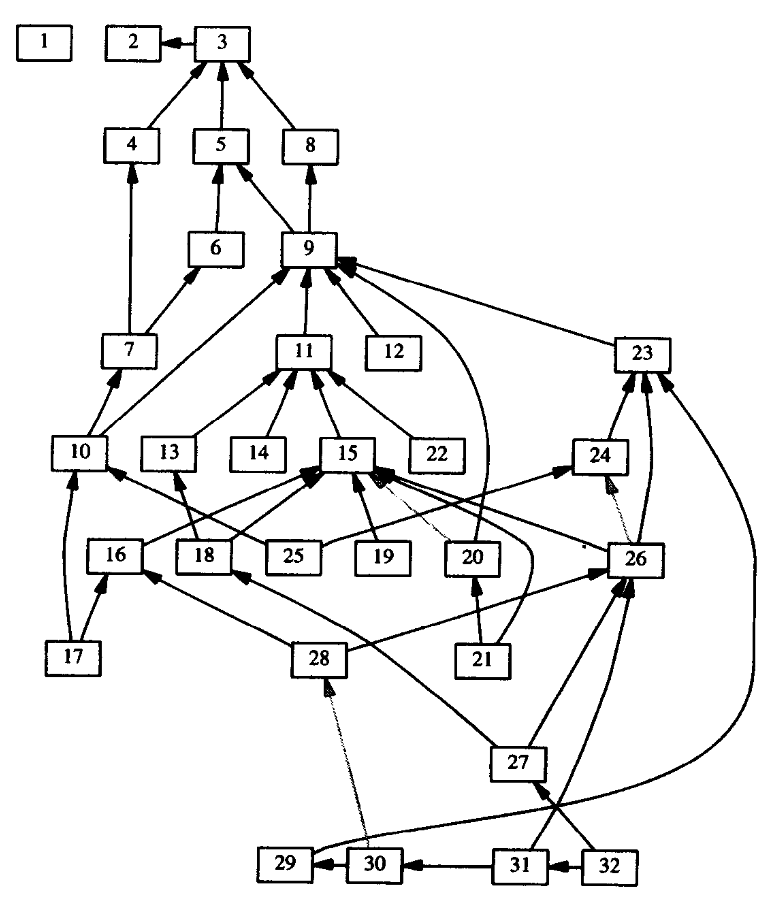


图 P.1　章节之间的依赖关系

本书对每个语言特征都采用统一的处理模式。先举一个例子,然后是形式定义;基本性质的证明,如类型安全性;通常另一章对理论进一步研究,得到类型检查算法及其可靠性,完备性

和终止性的证明;最后(在另一章中)给出这些算法的 OCaml 程序具体实现。

本书中的例子主要是对面向对象程序进行特征分析和设计的。有 4 个实例学习章节,详细地讨论了不同的内容:常规命令性对象和类的一个简单模型(参见第 18 章);基于 Java 的一个核心演算(参见第 19 章);用囿量词对命令性对象的一个更细致说明(参见第 27 章),以及用存在类型处理纯函数式的系统 $F_{<:}^{\omega}$ 中的对象和类(参见第 32 章)。

为使本书适合一学期的教材,必须忽略掉一些有意义和重要的论题。在本书中,指称和公理语义完全忽略掉了;类型系统与逻辑之间的联系在几个地方提到但没有详细讨论。程序语言和类型系统的许多新的特征只是提及而已,如依赖类型,交叉类型,以及 Curry-Howard 对应;只用了少量的篇幅对这些特征做了介绍以供日后有兴趣的读者自己深入学习。最后,除了简单介绍 Java 形式的核心语言外(参见第 19 章),本书完全注重基于 lambda 演算的系统;但是,这样处理的概念和机制可以直接转换到像类型并发语言,类型汇编语言,以及特别对象演算的相关领域中。

要求的背景知识

本书不要求有程序语言理论的背景知识,但读者应该具有一定的数学基础——特别是学过大学课程的离散数学、算法和基础逻辑等。

读者应该至少熟悉一种高阶函数式程序语言(如 Scheme,ML,Haskell 等)、程序语言和编译器的基本概念(抽象语法,BNF 语法,求值,抽象机器等)。这些在许多大学教材中可以找到;作者特别推荐 Friedman, Wand 和 Haynes(2001)的"Essentials of Programming Languages"和 Scott(1999)的"Programming Language Pragmatics"。本书中有几章以面向对象语言,如 Java 为例来说明一些问题十分有效。

为使本书成为浓缩的一学期的高级教材,且为减轻大部分研究生的负担,书中没有将许多有趣且重要的问题包括进来,也完全忽略了指称语义与公理语义,但有许多关于这方面的好书,可供查阅,如果在本书中将它们包括进来,会违背本书尽量实现的初衷。

关于类型检查器的具体实现中的几个重要概念是用 OCaml 语言编写的,其中 OCaml(Objective Caml 的缩写)为 ML 语言的另一个俗称。在这几章中介绍一些 OCaml 的预备知识对理解代码是有帮助的,但不一定有这个必要;本书中只用到了语言的一小部分。且有些特征在刚开始出现时就解释过了。所以这几章与本书的其他部分编写方式截然不同,完全可以略过。

目前关于 OCaml 的最好教材是 Cousineau 和 Mauny(1998)撰写的。随 OCaml 发布的资料也可在网上查阅(http://caml.inria.fr 和 http://www.ocaml.org)。

熟悉 ML 另一种称法——Standard ML 的读者可以很容易地理解 OCaml 代码。关于 Standard ML 的教材有 Paulson(1996)和 Ullman(1997)。

课程安排

一个中级或高级研究生的教材(一个学期)可以覆盖本书的大部分内容。图 P.2 给出了在 Pennsylvania 大学博士生高级课程的一个教学大纲(一周两次 90 分钟课时,假定以快速的方式覆盖了程序语言理论的最小预备知识)。

对于大学或初级研究生教材,有几个方案可以选择。程序中类型系统的课程可以注重引人各种类型特征和说明它们如何使用的章节,而忽略了大部分元理论和实现的章节。另外,基本

理论和类型系统实现的课程可以选择前面的章节，大概掠过第 12 章（以及第 18 章和第 21 章），牺牲书后面的章节。较短课程可以选择依赖关系图 P.1 中的有关章节。

本书也是适合程序语言理论更一般的研究生课程的主要教材。这样一个课程可以花半个或三分之二个学期学习本书的主要部分，然后将剩下的时间用于学习一节基于 Milner(1999)的 pi 演算的并发理论，它介绍了 Hoare 逻辑和公理语义（如 Winskel,1993）或研究一下一些高级语言特征，如连续或模块系统。

课程	论题	阅读
1.	课程概论；历史；组织	1,(2)
2.	基础：语法，操作语义	3,4
3.	介绍 lambda 演算	5.1,5.2
4.	形式化 lambda 演算	5.3,6,7
5.	类型；简单类型的 lambda 演算	8,9,10
6.	简单扩展；导出类型	11
7.	更多的扩展	11
8.	规范化	12
9.	引用；异常	13,14
10.	子类型	15
11.	子类型的元理论	16,17
12.	指令性对象	18
13.	轻量级的 Java	19
14.	递归类型	20
15.	递归类型的元理论	21
16.	递归类型的元理论	21
17.	类型重构	22
18.	全称多态	23
19.	存在多态；ADT	24,(25)
20.	囿量词	26,27
21.	囿量词的元理论	28
22.	类型算子	29
23.	F_ω 的元理论	30
24.	高阶子类型	31
25.	纯函数式对象	32
26.	补充课程	

图 P.2　一个高级研究生课程的教学大纲样例

在本课程中，学期报告起了很大的作用，所以它可能会推延某些理论部分的学习（如规范化及某些元理论的章节），在学生选择报告的内容之前可以阅读大量的例子。

练习

大部分章节包括许多练习——一些需要动笔，一些包含所讨论的演算程序，以及关于这些演算的 ML 实现。每个练习的困难程度用下面的符号表示：

★	快速检查	30 秒到 5 分钟
★★	容易	≤1 小时
★★★	中等	≤3 小时
★★★★	具有挑战性	>3 小时

标记为★的练习是用于对重要概念的实时检查。建议读者在读后面的内容之前在此稍微停一会，思考一下这些问题。在每个章节中，可作为课外作业的练习标记为“推荐”。

附录 A 给出了大部分练习的答案。标记↛ 表示练习的答案没有在附录 A 中给出。

本书约定

大部分章节以一种散漫的形式引入某类型系统的特征，然后在一个或多个图表中将推论规则集中起来对系统进行形式化定义。为参考方便，这些定义通常不仅包括目前还在讨论的新特征和新规则，同时也给出构建完整演算所需要的其余规则。

本书的另一特色是在编写过程中所有的例子都经过了类型检查器的检查，原稿经过仔细检查，例子被抽取出，产生并编译出包含所讨论的特征的类型检查器，并将其用于检查这些例子，然后将检查结果插入到文本中。完成这一艰巨任务的系统称为 TinkerType，是由 Michael Levin 和作者本人(2001)开发的，由(美国)国家自然科学基金 CCR-9701826，"Principled Foundations for Programming with Objects"和 CCR-9912352，"Modular Type Systems"所资助。

Web 资源

与本书相关的网站是：

http://www.cis.upenn.edu/~bcpierce/tapl

这个网站中的资源包括本书的勘误表，课程计划建议，读者提供的附加材料，以及每章演算的实现(类型检查器和简单翻译器)集合。

这些实现提供了本书中的例子和检查练习答案的一个环境。它们也可以用于阅读和修改，已成功地由参加作者课程的学生用做小实现练习和大课程计划的基础。实现是用 OCaml 编写的。OCaml 编译器可以免费在 http://caml.inria.fr 中下载，并可安装在大部分平台中。

本网站还为读者提供了类型论坛(Types Forum)，一个 E-mail 列表涵盖类型系统及其应用的所有方面。列表确保低容量和高信噪比。存档和预定指令在 http://www.cis.upenn.edu/~bcpierce/types中。

致谢

读后感到受益匪浅的读者应感谢 4 位导师——Luca Cardelli，Bob Harper，Robin Milner，以及 John Reynolds。他们教给作者大部分关于程序语言和类型的知识。

其余的内容是通过合作所学到的；除了 Luca，Bob，Robin 和 John 之外，在这些研究中的合作者还包括 Martín Abadi，Gordon Plotkin，Randy Pollack，David N。Turner，Didier Rémy，Davide Sangiorgi，Adriana Compagnoni，Martin Hofmann，Giuseppe Castagna，Martin Steffen，Kim Bruce，Naoki Kobayashi，Haruo Hosoya，Atsushi Igarashi，Philip Wadler，Peter Buneman，Vladimir Gapeyev，Michael Levin，Peter Sewell，Jérôme Vouillon 和 Eijiro Sumii。这些合作不仅形成了作者理解的基础，也激发了极大的兴趣。

本书的结构和组织是通过与 Thorsten Altenkirch，Bob Harper，John Reynolds 教学讨论修改的，加上了如下人员的评论和提出的错误：Jim Alexander，Penny Anderson，Josh Berdine，Tony Bonner，John Tang Boyland，Dave Clarke，Diego Dainese，Olivier Danvy，Matthew Davis，Vladimir Gapeyev，Bob Harper，Erik Hilsdale，Haruo Hosoya，Atsushi Igarashi，Robert Irwin，Takayasu Ito，Assaf Kfoury，Michael Levin，Vassily Litvinov，Pablo López Olivas，Dave MacQueen，Narciso Marti-Oliet，Philippe Meunier，Robin Milner，Matti Nykänen，Gordon Plotkin，John Prevost，Fermín Reig，Didier Rémy，John Rey-

nolds, James Riely, Ohad Rodeh, Jürgen Schlegelmilch, Alan Schmitt, Andrew Schoonmaker, Olin Shivers, Perdita Stevens, Chris Stone, Eijiro Sumii, Val Tannen, Jérôme Vouillon 以及 Philip Wadler(如果不小心漏掉某个人,将非常抱歉)。Luca Cardelli, Roger Hindley, Dave MacQueen, John Reynolds, 以及 Jonathan Seldin 对某些混乱不清的历史观点的权威看法。

在 Indiana 大学(1997,1998)和 Pennsylvania 大学(1999,2000)的研究生讨论班的人员认真阅读了本书的较早版本,根据他们的反映和评价,作者将本书写成现在这个样子。Bob Prior 和 MIT 出版社在出版过程的许多阶段引导修改手稿。

程序的证明对数学的社会过程来说太无聊而作用不大。

——Richard DeMillo, Richard Lipton, Alan Perlis, 1979

因此不要依赖于社会过程来验证。

——David Dill, 1999

形式方法只有到了不理解它们的人们都能使用时才会产生重大影响。

——Tom Melham

目　　录

第二部分 简单类型

第三部分 子类型化

第四部分　递 归 类 型

第五部分 多 态

第六部分 高 阶 系 统

第1章　引　　论

1.1　计算机科学中的类型

现代软件工程应用广泛的形式方法来确保系统按照那些对其应有的行为进行隐式或显式的说明方式来实现。一方面是强有力的框架理论，像 Hoare 逻辑、代数说明语言、模态逻辑和指称语义；这些框架虽然能表述十分普遍且正确的性质，但难以使用，且对程序员也提出更高的要求。另一方面是能力适度的技术——适度到能够将自动检查器放到编译器、连接器或程序分析器中，这样能被不熟悉所基于的理论的程序员所使用。这种轻量级形式方法的一个典型实例就是模型检查器，作为芯片设计或通信协议这样的有限状态系统中搜索错误的工具。另一个越来越引起重视的实例是实时监测，使一个系统能动态地监测到它的部件是否处在说明所要求的状态中的技术。但至今最流行的和发展最完善的轻量级的形式方法是类型系统，这是本书的中心议题。

如大型社团共享许多术语一样，程序语言设计人员及实现人员对"类型系统"的定义包括了它的非正式用法，能做到这一点是十分不容易的事，尽管如此，该定义仍存在一些争议。下面是一个比较正确的定义：

> 类型系统是指一种根据所计算出值的种类对词语进行分类从而证明某程序行为不会发生的可行语法手段。

关于这个定义有几点评论。首先，这个定义确立类型系统作为程序的推理工具。这句话反映了本书对待程序语言中类型系统的态度。更一般地，类型系统（或类型理论）是指在逻辑、数学和哲学中更广泛的一类研究领域。这个意义下的类型系统最早形式化于 1900 年左右，是作为一种避免威胁数学基础的逻辑悖论，如 Russell 悖论（Russell，1902）的办法。在 20 世纪，类型成为逻辑，特别是证明论（参见 Gandy，1976 以及 Hindley，1997）中的标准工具，并渗透到哲学和科学的语言中。这一领域的主要里程碑包括 Russell 原始的类型分支理论（Whitehead 和 Russell，1910），Ramsey 简单类型理论（Ramsey，1925），Church 简单类型 lambda 演算的基础（1940），MartinLöf 构造性类型理论（1973，1984），以及 Berardi，Terlouw 和 Barendregt 的纯类型系统（Berardi，1988；Terlouw，1989；Barendregt，1992）。

即使在计算机科学里，有两种研究类型系统的重要分支。本书关注的是更实用的分支，因为它涉及程序语言的应用。偏于抽象的分支关注的是如何通过 Curry-Howard 对应（参见 9.4 节）将多种"纯类型 lambda 演算"与不同的逻辑联系在一起。两个分支采用类似的概念、符号和技术，但在方向上有着重要的差别。比如，类型 lambda 演算的研究通常关心的系统是每个良类型的计算是可终止的，但大部分程序语言为了保留像递归函数定义这样的特征而忽略了这个性质。

上述定义中另一个重点是强调项（语法词）的分类，即执行时根据项所计算的值的性质来对项分类。一个类型系统可以看做是用一个程序中项的静态计算来模拟项的执行时间行为

(此外,指派给项的类型通常是可复合计算的,即一个表达式的类型只依赖它的子表达式的类型)。

术语"静态"有时是有意加上的,如我们说一个"静态类型程序语言"是为了将这里的编译时间分析与某些语言中的动态或潜在类型区分开来,如语言 Scheme(Sussman 和 Steele,1975;Kelsey,Clinger 和 Rees,1998;Dybvig,1996),它的运行时类型标记是用来区分堆中不同种类的结构。术语"动态类型化"有点用词不当,应当换为"动态检查",但用法是按标准进行的。

静态的类型系统必然保守:它们能泛泛地证明某些坏的程序行为不会出现,但不能证明这些坏的程序行为的出现,并且它们有时还会拒绝在执行时间行为良好的程序。比如,一个像:

if < complex test > then 5 else < type error >

的程序被认为不是良类型的,即使 < complex test > 总是求值为 true,因为一个静态分析不能确定是这种情况。保守性和表达方式之间的矛盾是类型系统设计中一个基本事实。企图使更多的程序类型化(通过指派更精确的类型到程序中)就成为这个领域研究的主要动力。

相关的一点是嵌入在大部分类型系统中的相对直接的分析,不能阻止任意意想不到的程序行为;他们只能保证良类型程序没有某类错误行为。比如,大部分类型系统能静态地检查初始算术运算的变量总是数字,在一个方法调用中的接受对象能提供所要求的方法,等等,但不能检查到除法运算中的第二个变量是否非零,或对数组的访问是否未越界。

给定语言中类型系统能消除的不良行为常常称为执行时间类型错误。记住这一点很重要:每种语言都存在这类行为,尽管不同语言执行时间错误的行为之间有着本质的重叠部分,原则上每个类型系统有着各自要阻止的行为定义。每个类型系统的安全性(或可靠性)必须根据自己的执行时间错误集合来判定。

类型分析发现的不良行为不限于低层错误,如调用一个不存在的方法:类型系统也用来加强高层模块性质,保护用户定义的抽象完整性。违反信息隐藏,如直接访问抽象数据值区域,将与一个整数处理为一个指针,从而使机器崩溃一样,都是执行时间错误。

类型检查器专门置于编译器或连接器中。这意味着它们必须能自动完成它们的工作,不需要人工干预或程序员的介入——即它们必须体现计算可行性分析。尽管如此,仍然有许多地方需要程序员的引导,即在程序中加入明显的类型注释。通常,这些注释会相对少些,使得程序更容易写和读。但原则上,程序符合某些规范的一个完整证明可以在类型的注释中编码;这种情况下,类型检查器将相应地变成一个证明检查器。如扩展静态检查技术(Detlefs,Leino,Nelson 和 Saxe,1998)是类型系统用来对一个全面程序进行验证的方法:只需通过少量合理的程序注释来帮助实现某些正确性的全自动检查。

由于同样的原因,我们最感兴趣的不是自动实现方面,而是能得出有效类型检查算法的方法。但是,什么是有效的仍是争论的焦点。即使广泛使用的类型系统,如 ML(Damas 和 Milner,1982),可能在病态情况下(Henglein 和 Mairson,1991)需要大量的类型检查时间。也有一些语言,它们的类型检查或类型重构问题是不可判定的,但对那些在"大部分实际情况下"可以很快停止的程序可以找出有效的算法(如 Pierce 和 Turner,2000;Nadathur 和 Milner,1988;Pfenning,1994)。

1.2 类型系统的优点

侦测错误

静态类型检查最明显的优点是它总能早早地侦测程序错误。早期侦测出的错误可以立刻定位,而不会隐藏在代码中待以后发现(即当程序员忙于其他事情或甚至在程序被配置以后)。此外,在事情开始变糟之前错误的作用不是很明显的情况下,类型检查时发现的错误比在执行时间发现的错误更精确。

在实际中,静态类型检查可暴露多得令人惊讶的错误。使用丰富类型语言的程序员常常感到他们的程序一旦通过类型检查器就会马上运行,在他们还来不及想下一步怎么做时,这种情况已经多次出现了。一个可能的解释是,这不但是平凡的人为事故(如忘记在求平方根之前将串转换为一个数),而且是更深的概念性错误(如忽略了一个复杂情况分析中的一个边界条件,或混淆一个科学计算中的单位),都会通过类型不一致体现出来。效果如何依赖于类型系统的表达能力和具体的程序:处理各种数据结构的程序(如编译器这样的符号处理应用)比起只涉及很少简单类型的程序,如科学应用中的数值计算[尽管如此,支持维度分析(Kennedy,1994)的改进类型系统也可以非常有用],常常更需要类型检查。

从类型系统得到的最大好处包括将注意力放在程序员的方面,以及可以充分利用语言提供的一切工具;比如,一个将所有的数据结构编译为列表的复杂程序,无法获得与将每个数据结构定义为不同数据类型或抽象类型的程序从编译器得到的帮助一样多。表达能力强的类型系统提供多种用类型编码结构信息的"技巧"。

对某些程序,一个类型检查器也可能成为一个无价的维护工具。比如,当程序员想改变复杂数据结构定义时不需要手工在一个大程序中查找所有出现这个结构的位置。一旦数据类型改变,所有这些位置都将变成类型不一致,这时能执行编译器将类型检查失效的位置列举出来。

抽象

类型系统支持程序设计的另一个重要方法是强化规范编程。特别是,在大型软件集成的环境中,类型系统是用来将大系统的组件打包和捆在一起的模块语言的主干。类型出现在模块的接口(及相关的结构,如类)中;的确,一个接口本身能看做是"一个模块类型",提供模块大概完成的功能——一种实现者与用户之间的部分契约。

用带清晰接口的模块来结构化大型系统会产生一个更为抽象的设计形式,其中接口的设计和讨论与最终的实现方式无关。对接口思考得越抽象越有利于设计。

文档

读程序时类型也有用。在程序头和模块接口中的类型声明形成一个文档,对程序的行为给出了有用提示。此外,不像嵌入在注释中的描述,文档的这种形式不会过时,因为在编译器的每次执行时都要检查它。类型的这个作用对模块基调特别重要。

语言安全性

遗憾的是,术语“安全语言”比起“类型系统”有着更多的争议。尽管人们都认为是按照自己的想法去理解的,但是实际上语言安全性概念的形成很大程度上受到了其所属的语言团体的影响。非形式地,安全语言可以定义为在编程时不可能危害到自身语言。

将这个说法再深入一点,我们可以说一个安全的语言是保护它自己的抽象的语言。每个高层语言提供机器服务的抽象。安全性是指语言具备保证这些抽象,以及程序员用语言的定义工具引入高层抽象的完整性的能力。比如,一个语言可以提供数组,具有访问和更新操作,作为基础内存抽象。一个使用这个语言的程序员期望数组只用更新操作才能改变——而不必接在其他数据结构末尾修改。类似地,程序员期望语义范围的变量只可以在它们的语义范围内访问,调用堆栈的行为真正地像一个堆栈等。在一个安全语言中,这样的抽象可以抽象地使用;在一个不安全的语言中则不能,为了完全理解一个程序怎样(错误)操作,必须记住所有的低层细节,如内存中数据结构的布局,以及由编译器分配的次序等。总之,不安全语言写出的程序不但可能破坏自己的数据结构,而且可能破坏那些执行时间系统的数据结构;在这种情况下的结果完全无法确定了。

语言安全性与静态类型安全性不是一回事。语言安全性可以由静态检查来保证,也可以由执行时间检查来保证,因为执行时间检查在无意义操作企图执行时会捕获这些操作,然后停止程序或提升一个异常。比如,Scheme 是一个安全语言,即使它没有静态类型系统。

反之,不安全的语言常常提供“最佳效果”的静态类型检查器,来帮助程序员消除至少是最明显的错误,但根据定义,这样的语言本身也不是类型安全的,因为它们一般不能保证良类型程序是良行为的——这些语言的类型检查器能提示会出现哪些执行时间类型错误(显然比提不出要好),但不能证明它们不会出现。

	静态检查	动态检查
安全	ML, Haskell, Java 等	Lisp, Scheme, Perl, Postscript 等
不安全	C, C++ 等	

在这个表中的右下角是空的,因为一旦检查工具用来可以加强大部分操作在执行时间的安全性,将没有附加的代价用于检查所有的操作(实际上,存在几个动态检查的语言,如带有最小操作系统的微型计算机的 BASIC 语言,提供对任意内存位置进行读写操作的低层原语,一旦错误地使用就会破坏执行时间系统的完整性)。

通常不能仅依靠静态类型化来保证运行时间的安全。例如,上表中所列的所有安全语言实际上都动态地执行数组越界检查[①]。类似地,静态检查语言有时也会选择采用一些操作(如 Java 中的向下转型算子,参见 15.5 节),这些操作中的类型检查规则实际上并不可靠,所以还需要在使用每种结构时进行动态检查以保证语言的安全性。

---

① 数组越界检查的静态消除是类型系统设计者的一个长期目标。原则上,大家都能理解有必要增加一些机制(基于依赖类型,参见 30.5 节),但如何把这些机制统一起来,使之能达到表达能力强,类型检查可预言性强和可行性好,以及程序注释的复杂度小,等等,都将是一个巨大的挑战。此领域的一些最新的进展可参见 Xi 和 Pfenning(1998, 1999)等人的著作。

语言安全性很少是绝对的。安全语言常常给程序员提供“安全舱口”，比如外部函数调用其他可能用不安全语言写的代码。的确，这样的安全舱口有时在语言内部以受限的形式提供，如 OCaml 的 Obj.magic(Leroy,2000)，在 Standard ML 的 New Jersey 实现中的 Unsafe.cast，等等。Modula-3(Cardelli 等,1989;Nelson,1991)和 C # (Wille,2000)提供一个“不安全的子语言”，用于实现低级执行时间完成的功能，如垃圾收集。这个子语言的特点可能只能用于明显注明 unsafe 的模块。

Cardelli(1996)明确提出关于语言安全的一个不同观点，区别所谓的捕获的和未捕获的执行时间错误。一个捕获错误引起一个计算立即停止(或提升一个异常，这个异常可以在程序内部彻底处理)，而未捕获的错误可以允许计算继续(至少持续一会)进行。一个未捕获错误会造成如 C 语言中访问超过数组最大下标的数据。在这个观点下，一个安全语言是可以防止出现在执行时间未捕获错误的语言。

还有一种观点关注可移植性;其宗旨是:一个安全语言完全是由它的程序员手册所定义的。假设一种语言的定义是程序员为了预测语言中所有程序应实现的功能而想出的概念集合。那么，一个像 C 语言的手册不构成这个定义，因为某些程序的行为(如包含未经检查的数组访问或指针算术)在不知道一个具体的 C 编译器如何配置内存中数据结构之前，是不能被预测的，并且同样的程序由不同编译器执行时可以有不同的行为。相比之下，Java，Scheme 以及 ML 的手册说明了(在不同严格程度下)语言中所有程序的精确行为。一个良类型程序在这些语言的正确执行下将产生相同的结果。

有效性

计算机科学中的第一个类型系统，是 20 世纪 50 年代的 FORTRAN(Backus,1981)语言，通过区别整数值算术表达式和实数值算术表达式来改善数值计算的有效性，这使得编译器可以用不同的表示形成，生成合适的原始运算机器指令。在安全语言中，若消除许多保证安全性(通过静态证明它们总是满足的)的动态检查，可以进一步改善有效性。今天，大部分高性能编译器主要依赖在优化和原码生成阶段由类型检查器收集的信息。即使没有类型系统的语言编译器，本质上也在努力获得这些类型信息。

类型信息还能在其他令人意想不到的地方改善有效性。比如，最近证明:在并行科学程序中不但原码生成决策，而且指针表示方式可以用类型分析生成的信息得到改善。Titanium 语言(Yelick 等,1998)所用的类型推理技术分析指针的辖域，能够做出比程序员手工调试程序更好的决定。ML Kit 编译器用一个强有力的区域推论算法(Gifford,Jouvelot,Lucassen 和 Sheldon,1987;Jouvelot 和 Gifford,1991;Talpin 和 Jouvelot,1992;Tofte 和 Talpin,1994,1997;Tofte 和 Birkedal,1998)来替代大部分(在有些程序中是所有的)基于堆栈的内存管理方式的垃圾收集。

进一步应用

除了在程序和语言设计中使用之外，类型系统还以许多特殊的方式应用于计算机科学和相关领域(我们将列举几个)。

类型系统应用的一个重要领域是计算机和网络安全。静态类型化是 Java 和 Jini 的网络设备的即插即用体系(Arnold 等,1999)的安全模型的核心，并且是携带证明代码的一个关键授权技术(Necula 和 Lee,1996,1998;Necula,1997)。同时，许多安全领域的基础性想法在程序语言

的环境中重新研究，而常常采用的方法是类型分析（如 Abadi，Banerjee，Heintze 和 Riecke，1999；Abadi，1999；Leroy 和 Rouaix，1998 等）。反之，将程序语言理论直接应用于在安全领域也是目前的研究趋势（Abadi，1999；Sumii 和 Pierce，2001）。

除编译器之外，类型检查和推论算法还会出现在许多程序分析的工具中。比如，AnnoDomini，一个 Cobol 程序的千年虫转换工具，是基于一个 ML 形式的类型推论引擎（Eidorff 等，1999）。类型推论技术也用于别名分析（O'Callahan 和 Jackson，1997）和异常分析（Leroy 和 Pessaux，2000）。

在自动定理证明中，类型系统（通常基于依赖类型的系统）用来表示逻辑命题和证明。几个常用的证明辅助器，包括 Nuprl（Constable 等，1986），Lego（Luo 和 Pollack，1992；Pollack，1994），Coq（Barras 等，1997），以及 Alf（Magnusson 和 Nordström，1994），是直接基于类型理论的。Constable（1998）和 Pfenning（1999）讨论了这些系统的历史。

在数据库领域，随着文档类型定义形成（XML 1998）和其他描述 XML 的数据结构的模式（如 XML-Schema 标准[XS 2000]）中“Web 元数据”的发展，人们对类型系统的兴趣越来越浓。查询和处理 XML 的新语言提供了直接基于这些模式语言的强大的静态类型系统（Hosoya 和 Pierce，2000；Hosoya，Vouillon 和 Pierce，2001；Hosoya 和 Pierce，2001；Relax，2000；Shields，2001）。

类型系统还应用于一个完全不同的领域——计算语言学领域，其中类型 lambda 演算形成了形式体系，如范畴语法（van Benthem，1995；van Benthem 和 Meulen，1997；Ranta，1995；等）的基础。

1.3 类型系统和语言设计

设计语言若没有考虑到类型检查，然后再将类型系统补充进来会有很多的困难；所以理想的做法是，语言设计与类型系统设计并行进行。

其中一个因素是不含类型系统的语言（即使是安全的，可以动态检查的语言）倾向于增加某些特征或采用某些编程习惯，从而使类型检查变得困难或不可行。的确，在类型化的语言中，在考虑设计的方方面面时类型系统本身常常看做是设计的基础和组织原则。

另一个因素是与无类型语言做比较，类型语言的具体语法更复杂，因为必须考虑类型注释。而在类型化语言中，当所有问题一起考虑时，设计一个清晰且易理解的语法会变得更容易。

类型应该是一个程序语言的不可或缺的部分，这个断言应该与程序员必须在哪里写类型注释，以及这些注释在哪里由编译器推导的问题分开来理解。一个设计良好的静态类型语言绝不要求写出大量需要由程序员维护的类型信息。关于多少类型信息才算太多这个问题一直都有争论。ML 系列的语言设计者努力使注释最少，并用类型推论方法恢复必要的信息。C 系列的语言（包括 Java），则采用了稍冗长的形式。

1.4 历史概要

在计算机科学中，最早的类型系统用来对数字的整数和浮点表示（如在 FORTRAN 中）做简单的区分。在 20 世纪 50 年代后期和 20 世纪 60 年代早期，这种分类扩展到了结构数据（记录数组等）和高阶函数。在 20 世纪 70 年代，引入了几个更丰富的概念（参数化多态，抽象数据

类型,模块系统和子类型),并且类型系统作为一个领域独立地出现了。同时,计算机科学家开始意识到程序语言中类型系统与数理逻辑中研究的类型系统之间的联系,开始了两方面的交叉研究。

图1.1给出了计算机科学中类型系统历史的一些重点(也许并不完整)年表。右边栏中列出的索引可以在参考文献中找到。

19世纪80年代	形式逻辑的起源	Frege(1879)
20世纪初	数学的形式化.	Whitehead 和 Russell (1910)
20世纪30年代	无类型演算	Church(1941)
20世纪40年代	简单类型化演算	Church(1940),Curry 和 Feys(1958)
20世纪50年代	FORTRAN	Backus(1981)
	Algol 60	Naur 等(1963)
20世纪60年代	Automath 计划	de Bruijn(1980)
	Simula	Birtwistle 等(1979)
	Curry-Howard 对应	Howard(1980)
	Algol 68	(van Wijngaarden 等(1975)
20世纪70年代	Pascal	Wirth(1971)
	Martin Löf 类型理论	Martin-Löf(1973,1982)
	系统 F, F^{ω}	Girard(1972)
	多态 lambda 演算	Reynolds(1974)
	CLU	Liskov 等(1981)
	多态类型推理	Milner(1978),Damas 和 Milner(1982)
	ML	Gordon, Milner 和 Wadsworth(1979)
	交类型	Coppo 和 Dezani(1978)
		Coppo, Dezani 和 Sallé(1979),Pottinger(1980)
20世纪80年代	NuPRL 计划	Constable 等(1986)
	子类型	Reynolds(1980),Cardelli(1984),Mitchell(1984a)
	作为存在类型的 ADT	Mitchell 和 Plotkin(1988)
	构造演算	Coquand(1985),Coquand 和 Huet(1988)
	线性逻辑	Girard(1987),Girard 等(1989)
	囿量词	Cardelli 和 Wegner(1985)
		Curien 和 Ghelli(1992),Cardelli 等(1994)
	Edinburgh 逻辑基础框架	Harper, Honsell 和 Plotkin(1992)
	Forsythe	Reynolds(1988)
	纯类型系统	Terlouw(1989),Berardi(1988),Barendregt(1991)
	依赖类型和模块化	Burstall 和 Lampson(1984),MacQueen(1986)
	Quest	Cardelli(1991)
	效果系统	Gifford 等(1987),Talpin 和 Jouvelot(1992)
	行变量,可扩展记录	Wand(1987),Rémy(1989)
		Cardelli 和 Mitchell(1991)
20世纪90年代	高阶子类型	Cardelli(1990),Cardelli 和 Longo(1991)
	类型化中间语言	Tarditi, Morrisett 等(1996)
	对象演算	Abadi 和 Cardelli(1996)
	透明类型和模块化	Harper 和 Lillibridge(1994),Leroy(1994)
	类型化汇编语言	Morrisett 等(1998)

图1.1 类型在计算机科学和逻辑中的历史概要

1.5 相关阅读

本来是希望读者只阅读本书就能掌握其中的内容,但这反而会使读者更难以理解;因为对于一本书来说,这个领域太大了,可以从太多的角度来考虑。所以本节列出几本好的参考书籍。

Cardelli(1996)和 Mitchell(1990b)的手册对这一领域做了简要介绍。Barendregt(1992)的文章更注重数学方面。Mitchell 关于"Foundations for Programming Languages"(1996)的教材涵盖了基本 lambda 演算,类型系统和语义等许多方面。重点在语义而不是在实现问题。Reynolds 的"Theories of Programming Languages"(1998b),是一本程序语言理论的研究生教材,包括了多态、子类型和交叉类型。Schmidt(1994)的"The Structure of Typed Programming Languages"研究了在语言设计的上下文中的类型系统的核心概念,包括几章关于约定性指令语言。Hindley 的专著"Basic Simple Type Theory"(1997)概要介绍了简单类型 lambda 演算和密切与之相关的系统。它更注重覆盖的深度而不是广度。

Abadi 和 Cardelli 的"A Theory of Objects"讨论了几个与本书相同的问题,但重点不在实现方面,而是这些思想在面向对象程序的基础处理中的应用。Kim Bruce 的"Foundations of Object-Oriented Languages: Types and Semantics"(2002)涵盖了类似的方面。Palsberg 和 Schwartzbach (1994),以及 Castagna(1997)中也给出了面向对象类型系统的基础介绍。

Gunter(1992),Winskel(1993)以及 Mitchell(1996)的教材深入讨论了无类型和类型语言的语义基础。Hennessy(1990)讨论了操作语义。有许多图书都介绍了范畴论数学框架中的类型语义基础,包括 Jacobs(1999),Asperti 和 Longo(1991),以及 Crole(1994)的著作."Basic Category Theory for Computer Scientists"(Pierce,1991a)中介绍了这方面的初步知识。

Girard,Lafont 和 Taylor 的"Proofs and Types"(1989)讨论类型系统的逻辑方面(Curry-Howard 对应,等等),它包括系统的描述和线性逻辑介绍性的内容。Pfenning 的"Computation and Deduction"(2001)进一步讨论了类型和逻辑之间的联系。Thompson 的"Type Theory and Functional Programming"(1991)和 Turner 的"Constructive Foundations for Functional Languages"(1991)从一个逻辑角度看函数式程序(在 Haskell 或 Miranda 的"纯函数式程序"的意义下)和构造类型论之间的联系。Goubault-Larrecq 和 Mackie 的"Proof Theory and Automated Deduction"(1997)讨论了几个与证明论相关的主题。在 Constable(1998),Wadler(2000),Huet(1990)和 Pfenning(1999)的书中,Laan 的博士论文(1997),Grattan-Guinness(2201),以及 Sommaruga(2000)的书中都详细地描述了逻辑学及哲学中类型的发展史。

在说明程序语言的类型可靠性方面,需要进行大量的详细分析来避免产生错误和令人尴尬的提法。因此,对类型系统的分类,描述和研究已成为一种形式方法。

——Luca Cardelli(1996)

第2章 数学基础

在进入正题之前,我们需要介绍一些基本概念和几个基本数学事实。大部分读者可以跳过这一章,需要时再回来翻阅。

2.1 集合、关系和函数

2.1.1 定义:我们用标准的符号表示集合:大括号用于列出一个集合的元素($\{\cdots\}$),或者用包含的方式从一个集合构造另一个集合($\{x \in S \mid \cdots\}$),$\varnothing$表示空集,$S \setminus T$ 表示 S 和 T 的集合差(不在 T 中而在 S 中的元素集合)。一个集合 S 的长度记为$|S|$。S 的幂集,即 S 的所有子集的集合,记为$\mathcal{P}(S)$。

2.1.2 定义:自然数集$\{0,1,2,3,4,5,\cdots\}$记为$\mathbb{N}$。如果它的元素可以与自然数一一对应,则一个集合称为可数的。

2.1.3 定义:集合 $S_1, S_2, \cdots, S_n$ 上的一个 n 元关系是 S_1 到 S_n 的 n 元组的集合 $R \subseteq S_1 \times S_2 \times \cdots \times S_n$。我们说元素 $s_1 \in S_1, \cdots, s_n \in S_n$ 是 R 关联的,如果$(s_1, \cdots, s_n)$是 R 的元素。

2.1.4 定义:集合 S 上的一个一元关系称为 S 上的一个谓词。(如果 $s \in P$),称 P 对元素 $s \in S$ 是真的。为了强调这一点,我们常常用 $P(s)$表示 $s \in P$,将 P 看成是将 S 的元素映射为真假值的一个函数。

2.1.5 定义:集合 S, T 上的关系 R 称为二元关系。常常用 $s\ R\ t$ 表示$(s,t) \in R$。当 S, T 为相同集合 U 时,我们称 R 为 U 上的一个二元关系。

2.1.6 定义:为了可读性,三元或三元以上的关系常常用混合记法,其中用关系将关联的元素分开。比如,在第 9 章中简单类型 lambda 演算中类型关系,我们用 $\Gamma \vdash \mathrm{s}:\mathrm{T}$ 表示"三元组$(\Gamma, \mathrm{s}, \mathrm{T})$是类型关系关联的"。

2.1.7 定义:S, T 上的一个关系 R 的定义域,记为 $dom(R)$,是元素 $s \in S$ 的集合,使得对某个 t,有$(s,t) \in R$。R 的值域,记为 $range(R)$,是元素 $t \in T$ 的集合,使得对某个 s,有$(s,t) \in R$。

2.1.8 定义:S, T 上的一个关系称为 S 到 T 的一个部分函数 R,如果,当$(s,t_1) \in R$ 且$(s,t_2) \in R$,则有 $t_1 = t_2$。如果 $dom(R) = S$,则 R 称为一个 S 到 T 的全函数(简称为函数)。

2.1.9 定义:一个 S 到 T 的部分函数 R,对元素 $s \in S$,若有 $s \in dom(R)$则称 R 为有定义的。否则称为无定义(发散的)。我们用 $f(x)\uparrow$ 或 $f(x) = \uparrow$ 表示 f 在 x 上无定义,用 $f(x)\downarrow$ 表示f 在 x 上有定义。

在讨论实现的章节中,我们将定义在某输入下失败的函数(如图 22.2 所示)。区别失败(一个合法可见的结果)和发散是重要的;一个可能失败的函数可以是部分函数(即在某些元素

$s\in S$ 上它可能发散)也可以是全函数(它总是输出一个结果或者明确的失败)。的确,我们将常常需要证明全函数性。用 $f(x)=fail$ 表示 f 在输入 x 下输出一个失败的结果。

形式地,S 到 T 的可能失败的函数实际上是 S 到 $T\cup\{fail\}$ 的一个函数,其中我们假定 $fail$ 不属于 T。

2.1.10 定义:假设 R 是一个集合 S 上的二元关系,P 是 S 上的一个谓词。如果 $s\,R\,s'$ 和 $P(s)$ 满足,则 $P(s')$ 也满足,那么说 P 在 R 下是保持的。

2.2 有序集合

2.2.1 定义:如果 R 关联 S 的每个元素到其自身,即对所有的 $s\in S$,$s\,R\,s$(或 $(s,s)\in R$),则一个集合 S 上的二元关系 R 是自反的。R 是对称的,如果对所有的 s,有 $t\in S$,$s\,R\,t$ 蕴涵 $t\,R\,s$。如果 $s\,R\,t$ 和 $t\,R\,u$ 蕴涵 $s\,R\,u$,则 R 是传递的。如果 $s\,R\,t$ 和 $t\,R\,s$ 蕴涵 $s=t$,则 R 是反对称的。

2.2.2 定义:一个集合 S 上的自反,传递关系 R 则称为 S 上的一个前序(也称伪序)(当我们说一个前序集合 S 时,总是指 S 上某个特殊的前序 R)。前序通常用符号 $\leqslant$ 或 $\sqsubseteq$ 表示。我们用 $s<t$(s 严格小于 t)表示 $s\leqslant t\wedge s\neq t$。

一个(在集合 S 上的)前序,如果是反对称的,则称为 S 上的一个偏序。如果对 S 中的每个 s 和 t,或者 $s\leqslant t$ 或者 $t\leqslant s$,则一个偏序 $\leqslant$ 称为一个全序。

2.2.3 定义:假设 $\leqslant$ 是一个集合 S 上的偏序,且 s,t 是 S 的元素。一个元素 $j\in S$ 称为 s 和 t 的合(最小上界),如果:

1. $s\leqslant j$,$t\leqslant j$,且
2. 对任何元素 $k\in S$ 满足 $s\leqslant k$ 且 $t\leqslant k$,则有 $j\leqslant k$。

类似地,一个元素 $m\in S$ 称为 s 和 t 的交(最大下界),如果:

1. $m\leqslant s$,$m\leqslant t$,且
2. 对任何元素 $n\in S$ 使得 $n\leqslant s$ 且 $n\leqslant t$,则有 $n\leqslant m$。

2.2.4 定义:一个集合 S 上自反,传递和对称的关系则称为 S 上的一个等价关系。

2.2.5 定义:假设 R 是集合 S 上一个二元关系。R 的自反闭包是包含 R 的最小的自反关系 R'("最小"是指如果 R'' 是某个其他包含 R 的自反关系,则有 $R'\subseteq R''$)。类似地,R 的"传递闭包"是包含 R 的最小的传递关系 R'。R 的传递闭包常常记为 R^+。R 的自反和传递闭包是包含 R 的最小的自反和传递关系,记为 R^*。

2.2.6 练习[★★ ↛]:假设给定集合 S 上一个关系 R,定义关系 R' 为:

$$R'=R\cup\{(s,s)\mid s\in S\}$$

即,R' 包含 R 中的所有的序对及 (s,s)。证明 R' 是 R 的自反闭包。

2.2.7 练习[★★ ↛]:下面是一个关系 R 的传递闭包的构造定义。首先,我们定义下列序对的序列:

$$R_0 = R$$

$$R_{i+1} = R_i \cup \{(s,u) \mid \text{对某个 } t, (s,t) \in R_i \text{ 且}(t,u) \in R_i\}$$

即,R_i 加上 R_i 中由序对一步传递得到的所有序对集合,得到新的 R_{i+1}。最后,定义 R^+ 为所有 R_i 的并:

$$R^+ = \bigcup_i R_i$$

证明 R^+ 是 R 的传递闭包,即它满足定义(2.2.5)中给出的条件。

2.2.8 练习[★★ ↛]:假设 R 是集合 S 上的一个二元关系,且 P 是 S 上在 R 下保持的一个谓词。证明 P 在 R^* 下也是保持的。

2.2.9 定义:假设在集合 S 上有一个前序≤。≤的一个降链是 S 的元素序列 $s_1, s_2, s_3, \cdots$ 使得序列的每个成员严格小于它的前驱:对每个 i, $s_{i+1} < s_i$[链可以是有限的,也可以是无限的,但我们常常对无限的链感兴趣,如定义(2.2.10)所示]。

2.2.10 定义:假设有集合 S 上的一个前序≤。我们称≤是良定的,如果它不包含无限的降链。比如,自然数的序,$0<1<2<3<\cdots$是良定的,但整数上的序,$\cdots<-3<-2<-1<0<1<2<3<\cdots$则不是。我们有时不特意提起≤,只说 S 是一个良定的集合。

2.3 序列

2.3.1 定义:一个序列即为元素的列,中间用逗号隔开。我们用逗号表示序列上的 cons 操作(加一个元素到序列的任意两端)和序列的添加(append)运算。比如,如果 a 是一个序列 3,2,1 且 b 是序列 5,6,则 0,a 表示序列 0,3,2,1,而 a,0 表示 3,2,1,0 且 b,a 表示 5,6,3,2,1(只要我们不讨论序列本身的顺序,逗号表示 cons 和 append 这两个运算不会导致混淆)。从 1 到 n 的自然数序列简记为 $1..n$(只用两个点)。我们$|a|$用表示序列 a 的长度。空序列记为•或空格。一个序列称为是另一个序列的置换,如果它们包含了相同的元素,只是次序不同。

2.4 归纳

如同在大部分计算机科学中一样,归纳证明在程序语言理论中是普适的证明方法。许多这样的证明是基于下列原理之一。

2.4.1 公理[自然数上一般的归纳原理]:假定 P 是自然数上一个谓词,则:

如果 $P(0)$

且对所有的 i,$P(i)$蕴涵 $P(i+1)$

则 $P(n)$对所有的 n 成立。

2.4.2 公理[自然数上的完全归纳原理]:假定 P 是自然数上一个谓词,则:

如果对每个自然数 n

　　假定 $P(i)$ 对所有的 $i<n$

　　我们能证明 $P(n)$

则 $P(n)$ 对所有的 n 成立。

2.4.3 定义:自然数序对上的字典序定义如下:$(m,n)\leqslant(m',n')$ 当且仅当或者 $m<m'$,或者 $m=m'$ 且 $n\leqslant n'$。

2.4.4 公理[字典序归纳原理]:假设 P 是自然数序对上一个谓词

如果对每个自然数序对 (m,n)

　　假定 $P(m',n')$ 对所有的 $(m',n')<(m,n)$

　　我们能证明 $P(m,n)$

则 $P(m,n)$ 对所有的 m,n 成立。

字典序归纳原理是嵌套归纳证明的基础,其中一个归纳证明的某个情况是由一个"内部归纳"证明给出的。这可以推广到自然数的三元组、四元组等上的字典序归纳(序对上的归纳是相当普遍的;有时可以见到有用的三元组上的归纳证明;三元组以上的归纳证明很少)。

第 3 章的定理(3.3.4)将引进归纳证明的另一种形式:称为结构归纳,这方法对树结构,如项或类型推导的证明特别有用。归纳推理的数学基础将在第 21 章中讨论,在那里我们将看到所有这些特别的归纳原理是更深刻思想的一个体现。

2.5 背景知识阅读

如果读者不熟悉这章的内容,可能需要阅读某些背景材料。有许多这方面的资料,其中 Winskel(1993)写的一本关于归纳法概念的书籍特别好。Davey 和 Priestley(1990)写的前几章很好地综述了有序集合。Halmos(1987)写了一本很好的关于集合论的入门书籍。

一个证明就是具有说服力的可重复实验。

——Jim Horning

第一部分　无类型系统

第3章　无类型算术表达式[①]

为了严格地讨论类型系统和它们的性质，首先需要形式地讨论程序语言的某些基础方面。特别是，我们需要一个清晰的、精确的和数学上可行的工具来表示和推理程序的语法和语义。

本章和下一章将给出讨论仅含自然数和布尔值的小型语言所需要的工具。这个语言非常平凡，但它是引入几个基本概念的有效载体，如抽象语法、归纳定义和证明、求值和执行时间错误的建模等。第5章至第7章将对更强的语言——无类型 lambda 演算进行同样的定义，而且在无类型 lambda 演算中必须考虑名称绑定和代换。第8章将开始研究类型系统，并回到本章的简单语言，用它来介绍静态类型的基本概念。第9章将这些概念推广到 lambda 演算。

3.1　导论

本章用到的语言包含几个语法形式：布尔值常量 true 和 false，条件表达式，数值常量 0，算术算子 succ（后继），pred（前驱）和一个测试操作 iszero，当运用于0时输出值 true，运用于其他数时输出值为 false。这些可以综述为下列语法：

t ::=	项：
true	常量真
false	常量假
if t then t else t	条件表达式
0	常量0
succ t	后继
pred t	前驱
iszero t	0测试

这个语法的（以及整个书中的）记法接近于标准 BNF 的记法（参见 Aho，Sethi 和 Ullman，1986）。第一行（t ::= ）说明我们定义项的集合，并且将用字母 t 表示项。下面的每一行给出项的一个语法形式。在符号 t 出现的每一点，我们可以对任何项进行代换（右边是其说明）。

这个语法右端中的符号 t 称为元变量。它是这种意义下的变量：一个可以用其他特殊的项来代换。“元”是指它不是对象语言（我们正在描述其语法的简单程序语言）中的变量，而是元语言中的变量，其中的概念已经描述过了（事实上，目前的对象语言不含变量；我们将在第5章中引入变量）。前缀“元”来自于元数学（逻辑学的分支之一），主要讨论数学和逻辑推理系统（包括程序语言）的数学性质。在元数学中有一个概念——“元理论”，是指可以组成某种特殊的逻辑系统（或者程序语言）真语句的集合以及对这些语句的研究。这样，在本书中的“子类型的元理论”这样的短语可以理解为“带子类型系统性质的形式研究”。

---

① 本章中讨论的系统是布尔和数的无类型演算（参见图3.2）。相关的 OCaml 实现，在 Web 库中称为 arith（在第4章中描述）。可通过网址 http://www.cis.upenn.edu/~bcpierce/tapl 下载并建立这个检查器。

在本书中，用元变量 t，s，u，r，及变式(variant)t_1，s′表示目前讨论的对象语言中的项；其他符号将用于表示由其他语法范畴中的表达式。在附录 B 中给出了所有关于元变量的约定。

目前，字“项”和“表达式”可以通用。从第 8 章开始，当讨论带附加语法范畴，如类型的演算时，我们将用“表达式”表示所有语法短语(包括项表达式，类型表达式和分类表达式等)，而“项”表示那些表示计算的短语(即这些短语可以取代元变量 t)。

在目前语言中一个程序就是由上述语法定义出来的一个项。下面是程序的例子及其求值的结果。为简单起见，我们用普通的阿拉伯数字来表示数，而数的正式表示为 succ 作用于 0 的嵌套形式。比如 succ(succ(succ(0)))记为 3。

```
    if false then 0 else 1;
  ▸ 1

    iszero (pred (succ 0));
  ▸ true
```

在本书中，符号“▸”用于表示例子中求值的结果(为简单起见，当结果明显或不重要时将省略)。在排版时，例子将由所讨论的形式系统相应的实现来自动处理(这里为 arith)；所显示的结果是实现的实际输出。

为了便于阅读，在例子中 succ，pred，iszero 的复合参数用括号括起来①。项的语法中没有提到括号，只给出了它们的抽象语法。当然，括号出现与否对目前讨论的简单语言来说没有差别：括号通常用来解决语法中的歧义性，但目前这个语法没有歧义性，每个记号序列至多有一种方法来分解为项。在第 5 章中将继续讨论括号和抽象语法。

求值的结果是某特殊简单形式的项：它们要么是布尔常量要么是数(嵌套使用 0 或 succ)。这样的项称为值，它们在项的求值次序的形式化中起着特殊的作用。

注意：项的语法可能形成某些看起来奇怪的项，如 succ true 和 if 0 then 0 else 0。我们将在以后讨论这样的项。的确，在某种意义下这正是我们对这个小语言感兴趣的原因，因为这些项正是我们要在一个类型系统去排除的一类无意义程序。

3.2 语法

这里有几种等价的方法来定义我们语言的语法。在上一节的语法中已经看到了一种。这个语法实际上是下面归纳定义的一种紧致的表示方式。

3.2.1 定义[项，归纳定义]：项的集合是最小的集合$\mathcal{T}$，其中满足：

1. $\{true, false, 0\} \subseteq \mathcal{T}$；
2. 如果 $t_1 \in \mathcal{T}$，则$\{succ\ t_1, pred\ t_1, iszero\ t_1\} \subseteq \mathcal{T}$；
3. 如果 $t_1 \in \mathcal{T}$，$t_2 \in \mathcal{T}$，且 $t_3 \in \mathcal{T}$，则 if t_1 then t_2 else $t_3 \in \mathcal{T}$。

① 事实上，这里用来处理本章中例子的实现(在本书的 Web 站点上称为 arith)确实需要在 succ，pred，iszero 的复合参数中插入括号，尽管它们在没有括号的情况下可以无歧义分解。这是为了与以后的演算，这些演算在函数应用中采用类似的语法方式。

由于归纳定义在程序语言的研究中是普适的,值得仔细讨论一下。第一个子句告诉我们3个简单的表达式是在$\mathcal{T}$中。第2个句子和第3个子句给出规则,由这些规则可以判定一个复合表达式是否是在$\mathcal{T}$中。最后,"$\mathcal{T}$是最小的"表示:除了3个子句要求的之外,$\mathcal{T}$不含其他元素。

像上一节中的语法一样,这个定义没有涉及使用括号来标记复合子项。形式地,我们实际上定义$\mathcal{T}$作为树的集合,而不是字符串的集合。在例子中使用括号正是为了说明书本上以线性形式表示的项与实际的树形式的项之间的对应关系。

另一种归纳定义项的方法是逻辑系统的"自然演泽形式"中经常使用的二维推导规则:

3.2.2 定义[项,用推导规则定义]:项的集合由下列规则定义:

$$\texttt{true} \in \mathcal{T} \qquad \texttt{false} \in \mathcal{T} \qquad \texttt{0} \in \mathcal{T}$$

$$\frac{\texttt{t}_1 \in \mathcal{T}}{\texttt{succ t}_1 \in \mathcal{T}} \qquad \frac{\texttt{t}_1 \in \mathcal{T}}{\texttt{pred t}_1 \in \mathcal{T}} \qquad \frac{\texttt{t}_1 \in \mathcal{T}}{\texttt{iszero t}_1 \in \mathcal{T}}$$

$$\frac{\texttt{t}_1 \in \mathcal{T} \quad \texttt{t}_2 \in \mathcal{T} \quad \texttt{t}_3 \in \mathcal{T}}{\texttt{if t}_1\texttt{ then t}_2\texttt{ else t}_3 \in \mathcal{T}}$$

前3个规则重述了定义(3.2.1)的第一个子句;下面4个规则等价于子句(2)和子句(3)。每个规则读为:"如果线上所列的前提成立,则可以推导出线下的结论"。这里我们常常不明确说明$\mathcal{T}$是满足这些规则的最小集合。

这里,需要提起两个术语。第一,没有前提的规则(像上述前3个规则)通常称为公理。在本书中,术语"推导规则"一般包括公理和带一个或多个前提的"真规则"。通常公理表示中不需要使用线,因为没有东西在线上。第二(严格地),我们所称的"推导规则"实际上是规则模式,因为它们的前提和结论可以含元变量。形式地,每个规则模式表示无限多个具体规则,通过用合适的语法范畴中的短语一致地代换元变量而得到具体规则,即在上述规则中,可用每个可能的项代换 t。

最后,还有一种定义项的方法,更具体地给出了产生$\mathcal{T}$中元素的步骤。

3.2.3 定义[项,具体定义]:对每个自然数 i,定义集合 S_i 为:

$$\begin{aligned} S_0 &= \varnothing \\ S_{i+1} &= \quad \{\texttt{true}, \texttt{false}, \texttt{0}\} \\ &\quad \cup \ \{\texttt{succ t}_1, \texttt{pred t}_1, \texttt{iszero t}_1 \mid \texttt{t}_1 \in S_i\} \\ &\quad \cup \ \{\texttt{if t}_1\texttt{ then t}_2\texttt{ else t}_3 \mid \texttt{t}_1, \texttt{t}_2, \texttt{t}_3 \in S_i\} \end{aligned}$$

最后,设:

$$S = \bigcup_i S_i$$

S_0 是空集;S_1 只包含常量;S_2 包含常量及由常量通过 succ,pred,iszero 或 if 其中一个生成的短语;S_3 包含 S_2 中的短语以及由 S_2 中的短语用一个 succ,pred,iszero 或 if 生成的短语,依次类推。S 为这样定义的所有短语,即由常量通过有限次使用算术和条件算子得到的短语集合。

3.2.4 练习[★★]:S_3 有多少个元素?

3.2.5 练习[★★]:证明集合 S_i 是可累积的,即对每个 i,有 $S_i \subseteq S_{i+1}$。

上面这些定义从不同的方面刻画了项的集合：定义(3.2.1)和定义(3.2.2)将这个集合简单地刻画为满足某些“封闭性质”的最小集合；定义(3.2.3)说明如何实际构造这个集合为一个序列极限。

最后，我们来验证这两种方法实际定义了相同的集合。给出的证明相当详细，为了说明它们之间如何等价。

3.2.6　命题：$\mathcal{T} = \mathcal{S}$。

证明：$\mathcal{T}$定义为满足一定条件的最小集合。因此只要证明(a)$\mathcal{S}$满足这些条件，并且(b)任何满足这些条件的集合包含$\mathcal{S}$(即$\mathcal{S}$是满足条件的最小集合)。

为证明(a)，必须验证定义(3.2.1)中每一个条件对$\mathcal{S}$是成立的。第一，因为$\mathcal{S}_1 = \{\text{true}, \text{false}, 0\}$，很清楚，常量是在$\mathcal{S}$中的。第二，如果$t_1 \in \mathcal{S}$，则(因为$\mathcal{S} = \bigcup_i \mathcal{S}_i$)必定存在某个$i$使得$t_1 \in \mathcal{S}_i$。但由$\mathcal{S}_{i+1}$的定义，必定有succ $t_1 \in \mathcal{S}_{i+1}$，因此，succ $t_1 \in \mathcal{S}$；类似地，可以证明：pred $t_1 \in \mathcal{S}$且iszero $t_1 \in \mathcal{S}$。第三，如果$t_1 \in \mathcal{S}$，$t_2 \in \mathcal{S}$，且$t_3 \in \mathcal{S}$，则由类似讨论，有if t_1 then t_2 else $t_3 \in \mathcal{S}$。

为证明(b)，假设某个集合$\mathcal{S}'$满足定义(3.2.1)中的3个条件。对i做完全归纳，可证明每个$\mathcal{S}_i \subseteq \mathcal{S}'$，因而，$\mathcal{S} \subseteq \mathcal{S}'$。

假设对所有$j < i$，有$\mathcal{S}_j \subseteq \mathcal{S}'$。必须证明$\mathcal{S}_i \subseteq \mathcal{S}'$，因为$\mathcal{S}_i$的定义有两个子句(对$i = 0$和$i > 0$)，有两个情况要考虑。如果$i = 0$，则$\mathcal{S}_i = \varnothing$；因此，$\varnothing \subseteq \mathcal{S}'$。否则，对某个$j$，$i = j + 1$。设t是$\mathcal{S}_{j+1}$的某个元素。由于$\mathcal{S}_{j+1}$定义为3个较小集合的并，t必定属于这3个集合的某一个。这样有3种可能性要考虑：(1)如果t是一个常量，则由条件(1)，$t \in \mathcal{S}'$；(2)如果对某个$t_1 \in \mathcal{S}_j$，t有形式succ t_1，pred t_1，或者iszero t_1，则由归纳假设，$t_1 \in \mathcal{S}'$，并且由条件(2)，有$t \in \mathcal{S}'$；(3)如果对某个$t_1, t_2, t_3 \in \mathcal{S}_i$，t有形式：if t_1 then t_2 else t_3，则由归纳假设，t_1, t_2, t_3均在$\mathcal{S}'$中，并由条件(3)，t也在$\mathcal{S}'$中。

这样，我们证明了每个$\mathcal{S}_i \subseteq \mathcal{S}'$。由$\mathcal{S}$的定义，$\mathcal{S}$是所有的$\mathcal{S}_i$的并集，因此，$\mathcal{S} \subseteq \mathcal{S}'$。这样命题证明完成。

值得注意的是，这个证明是对自然数做完全归纳，而不是大家更熟悉的“基本情况/归纳情况”的形式。对每个i，假设所要证明的谓词对严格小于i的自然数均成立，并且证明对i也成立。本质上，这里的每一步是一个归纳，惟一特别的是i的较小值集合，根据需要归纳假设，正好是空集。这样的评论同样适合于本书中大部分归纳证明，特别是“结构归纳”证明。

3.3　对项的归纳

命题(3.2.6)中项集$\mathcal{T}$明确提供了推理其元素的一个重要原则。如果$t \in \mathcal{T}$，则下面3条中必有一个为真：(1)t是一个常量；(2)t形为succ t_1，pred t_1或iszero t_1，其中t_1是某个更小的项；(3)t形为if t_1 then t_2 else t_3，其中t_1, t_2, t_3是更小的项。用这样的结论可以做两件事：给出项集合上函数归纳定义，并且给出项的性质的归纳证明。比如，下面是一个函数的简单归纳定义，其中函数是将每个项t映射到该项t中所出现的常量集合。

3.3.1　定义：出现在项t中常量集合，记为*Consts*(t)，定义为：

$$
\begin{aligned}
&\mathit{Consts}(\mathrm{true}) &&= \{\mathrm{true}\}\\
&\mathit{Consts}(\mathrm{false}) &&= \{\mathrm{false}\}\\
&\mathit{Consts}(0) &&= \{0\}\\
&\mathit{Consts}(\mathrm{succ}\ t_1) &&= \mathit{Consts}(t_1)\\
&\mathit{Consts}(\mathrm{pred}\ t_1) &&= \mathit{Consts}(t_1)\\
&\mathit{Consts}(\mathrm{iszero}\ t_1) &&= \mathit{Consts}(t_1)\\
&\mathit{Consts}(\mathrm{if}\ t_1\ \mathrm{then}\ t_2\ \mathrm{else}\ t_3) &&= \mathit{Consts}(t_1) \cup \mathit{Consts}(t_2) \cup \mathit{Consts}(t_3)
\end{aligned}
$$

可由归纳定义计算的另一个项的性质为项的长度。

3.3.2 定义:一个项 t 的长度,记为 *size*(t),定义如下所示:

$$
\begin{aligned}
&\mathit{size}(\mathrm{true}) &&= 1\\
&\mathit{size}(\mathrm{false}) &&= 1\\
&\mathit{size}(0) &&= 1\\
&\mathit{size}(\mathrm{succ}\ t_1) &&= \mathit{size}(t_1) + 1\\
&\mathit{size}(\mathrm{pred}\ t_1) &&= \mathit{size}(t_1) + 1\\
&\mathit{size}(\mathrm{iszero}\ t_1) &&= \mathit{size}(t_1) + 1\\
&\mathit{size}(\mathrm{if}\ t_1\ \mathrm{then}\ t_2\ \mathrm{else}\ t_3) &&= \mathit{size}(t_1) + \mathit{size}(t_2) + \mathit{size}(t_3) + 1
\end{aligned}
$$

即 t 的长度 *size*(t)是它的抽象语法树中节点的个数。类似地,一个项 t 的深度,记为 *depth*(t),定义如下所示:

$$
\begin{aligned}
&\mathit{depth}(\mathrm{true}) &&= 1\\
&\mathit{depth}(\mathrm{false}) &&= 1\\
&\mathit{depth}(0) &&= 1\\
&\mathit{depth}(\mathrm{succ}\ t_1) &&= \mathit{depth}(t_1) + 1\\
&\mathit{depth}(\mathrm{pred}\ t_1) &&= \mathit{depth}(t_1) + 1\\
&\mathit{depth}(\mathrm{iszero}\ t_1) &&= \mathit{depth}(t_1) + 1\\
&\mathit{depth}(\mathrm{if}\ t_1\ \mathrm{then}\ t_2\ \mathrm{else}\ t_3) &&= \max(\mathit{depth}(t_1), \mathit{depth}(t_2), \mathit{depth}(t_3)) + 1
\end{aligned}
$$

根据定义(3.2.3),等价地,*depth*(t)是满足 $t \in S_i$ 的最小 i。

下面归纳证明说明了项中常量个数与项的长度之间的关系(性质本身是完全明显的,我们所感兴趣的是归纳证明的形式,在以后还将经常采用这种形式)。

3.3.3 引理:一个项 t 中不同常量的个数不大于 t 的长度 *size*(t),即 $|\mathit{Consts}(t)| \leqslant \mathit{size}(t)$。

证明:对 t 的深度做归纳。假设对所有比 t 小的项结论成立,我们证明对 t 也成立。这里有 3 种情况要考虑:

情况:t 是一个常量

直接有:$|\mathit{Consts}(t)| = |\{t\}| = 1 = \mathit{size}(t)$。

情况:$t = \mathrm{succ}\ t_1, \mathrm{pred}\ t_1$,或 $\mathrm{iszero}\ t_1$

由归纳假设,$|\mathit{Consts}(t_1)| \leqslant \mathit{size}(t_1)$。可计算得到:$|\mathit{Consts}(t)| = |\mathit{Consts}(t_1)| \leqslant \mathit{size}(t_1) < \mathit{size}(t)$。

情况:$t = \mathrm{if}\ t_1\ \mathrm{then}\ t_2\ \mathrm{else}\ t_3$

由归纳假设,$|\mathit{Consts}(t_1)| \leqslant \mathit{size}(t_1)$,$|\mathit{Consts}(t_2)| \leqslant \mathit{size}(t_2)$,且 $|\mathit{Consts}(t_3)| \leqslant \mathit{size}(t_3)$,可计算得到:

$$
\begin{aligned}
|Consts(\mathrm{t})| &= |Consts(\mathrm{t}_1) \cup Consts(\mathrm{t}_2) \cup Consts(\mathrm{t}_3)| \\
&\leqslant |Consts(\mathrm{t}_1)| + |Consts(\mathrm{t}_2)| + |Consts(\mathrm{t}_3)| \\
&\leqslant size(\mathrm{t}_1) + size(\mathrm{t}_2) + size(\mathrm{t}_3) \\
&< size(\mathrm{t})
\end{aligned}
$$

这个证明的形式可以重述为一个一般推理原则。包括两个项的证明中常用到的类似原则。

3.3.4　定理[项上归纳原理]:假设 P 是项上的一个谓词。

对深度的归纳:如果对每个项 s
假设对所有使得 $depth(\mathrm{r}) < depth(\mathrm{s})$的项 r 有 $P(\mathrm{r})$,我们能证明 $P(\mathrm{s})$
则 $P(\mathrm{s})$对所有的 s 成立

对长度的归纳:如果对每个项 s
假设对所有使得 $size(\mathrm{r}) < size(\mathrm{s})$的项 r 有 $P(\mathrm{r})$,我们能证明 $P(\mathrm{s})$
则 $P(\mathrm{s})$对所有的 s 成立

结构归纳:如果对每个项 s
假设对所有 s 的直接子项 r,$P(\mathrm{r})$成立,我们能证明 $P(\mathrm{s})$
则 $P(\mathrm{s})$对所有的 s 成立。

证明:留做练习(★★)。
对项的深度或长度的归纳类似于自然数上完全归纳方法(2.4.2)。通常结构归纳对应于通常的自然数归纳原则(2.4.1),其中归纳要求由假设 $P(n)$证明 $P(n+1)$。
像自然数归纳的不同形式一样,项的归纳方法选择的是谁能得出更简单的结构。形式地,它们是可相互推导的。对于简单证明,使用对长度、对深度还是对结构做归纳对证明影响不大。由于形式问题,通常尽可能地采用结构归纳,因为它直接对项处理,避免使用自然数。
对项做归纳的大部分证明有着类似的结构。在归纳的每一步,我们给定一个项 t,假设对所有 t 的子项(或所有较小的项)有 P 成立,证明性质 P 对该项也成立。对 t 的每个可能形式(true, false, 条件, 0, 等等)分别进行考虑,证明在每种情况下 P 对 t 均成立。由于整个结构中归纳证明中惟一不同的部分是各种情况的细节处理,通常省略那些相同的部分,只写出不同部分的证明。

证明:对 t 做归纳

情况:t = true
…证明 $P(\mathrm{true})$…

情况:t = false
…证明 $P(\mathrm{false})$…

情况:$\mathrm{t} = \mathrm{if}\ \mathrm{t}_1\ \mathrm{then}\ \mathrm{t}_2\ \mathrm{else}\ \mathrm{t}_3$
…用 $P(\mathrm{t}_1)$, $P(\mathrm{t}_2)$且 $P(\mathrm{t}_3)$,证明 $P(\mathrm{if}\ \mathrm{t}_1\ \mathrm{then}\ \mathrm{t}_2\ \mathrm{else}\ \mathrm{t}_3)$…
(对其他语法形式也类似)

对于许多归纳讨论[包括定义(3.3.3)的证明],不必这么详细地写出证明:在基本情况(对没有子项的项 t)中 $P(t)$是直接成立的,而在归纳情况中运用归纳假设于 t 的子项,结合结果很显然地得到 $P(t)$。实际上,比起验证一个证明,读者更容易自己写出证明(记住归纳假设的同时,检查一下语法)。在这种情况下,直接写出“对 t 做归纳”就是一个完全的可接受的证明。

3.4 语义形式

严格地给出了语言的语法以后,下面需要给出对项如何求值的精确定义,即语言的语义。这里有 3 种基本方法进行语义形式化:

1. 操作语义通过定义一个简单的抽象机器来说明一个程序语言的行为。这个机器是“抽象的”,指它用语言的项作为机器状态,而不是某种低层的微处理器的指令集。对于简单的语言,机器的状态就是一个项,并且机器的行为由一个转换函数来定义,对每个状态,要么通过对项做一步简化给出下一个状态,要么声明机器已经停止。一个项 t 的语义可以看做是机器在将 t 作为初始状态时达到的最后状态①。
   有时给出一个语言的几个不同操作语义是有用的(其中一些较为抽象),机器状态类似于程序员写的项,另一些接近于实际语言翻译器或编译器所处理的结构。证明在执行一个程序时,不同机器的行为在某种意义下是相互对应的,等价于证明语言的一个实现的正确性。
2. 指称语义采用更为抽象的语义:一个项的语义不仅是机器状态的序列,而且是某个数学对象,如一个数或一个函数。一个语言的指称语义是由一个语义域的集合和将项映射到域中元素的解释函数所组成。为各种语言特征建模寻找所需的合适语义域是域论的研究目的,这是一个丰富的研究领域。
   指称语义的一个主要优势是它对求值的细节进行抽象,突出语言的本质概念。所选的语义定义域集合的性质可用来推导出关于程序行为推理的有力规则——这些规则用于证明两个程序有相同的行为,或一个程序的行为满足某个规范。最后,由语义定义域集合的性质,可以直接得出在一个语言中的有些事情(需要的或不需要的)是不可能发生的。
3. 公理语义对于这些规则采用更为直接的处理方式:不是首先定义程序的行为(给出某种操作语义或指称语义),然后推导出这个定义的规则,而是公理方法用规则本身作为语言的定义。一个项的语义就是关于这个项我们能证明的东西。
   公理方法的优美之处在于它关注程序推理的过程。正是这个思想定义了计算机科学中的不变式。

在 20 世纪 60 年代和 20 世纪 70 年代,操作语义通常被认为劣于其他两种方法,因为它对快速而粗糙地定义语言特征虽然有用,但不细致,且数学根基不牢。但在 20 世纪 80 年代,更抽象的方法开始不断地遇到技术上的困难②,这时,操作方法的简单性和灵活性越来越受到人

① 严格地讲,我们这里所描述的是所谓的操作语义的小步形式,有时称为结构操作语义(Plotkin,1981)。练习(3.5.17)引入另一种大步形式,有时称为自然语义(Kahn,1987),其中抽象机器的一个单个转换就可以将项求值为最终值。

② 指称语义的劣势在于对非确定性和并发性的处理;对于公理语义,劣势为过程。

们的关注，特别是这个领域的新进展，如 Plotkin 结构操作语义(1981)，Kahn 的自然语义(1987)，以及 Milner 对 CCS 的研究(1980;1989;1999)。Milner 的 CCS 引入了更细致的形式化并说明了在指称语义的环境中有多少研究有力的数学技术转化为操作装置。操作语义已成为一个充满活力的研究领域，常常是定义程序语言和研究其性质所选择的方法。本书将只使用操作语义。

3.5 求值

我们暂时不考虑数字，只讨论布尔表达式的操作语义。图 3.1 给出其定义，下面将详细讨论它的各个部分。

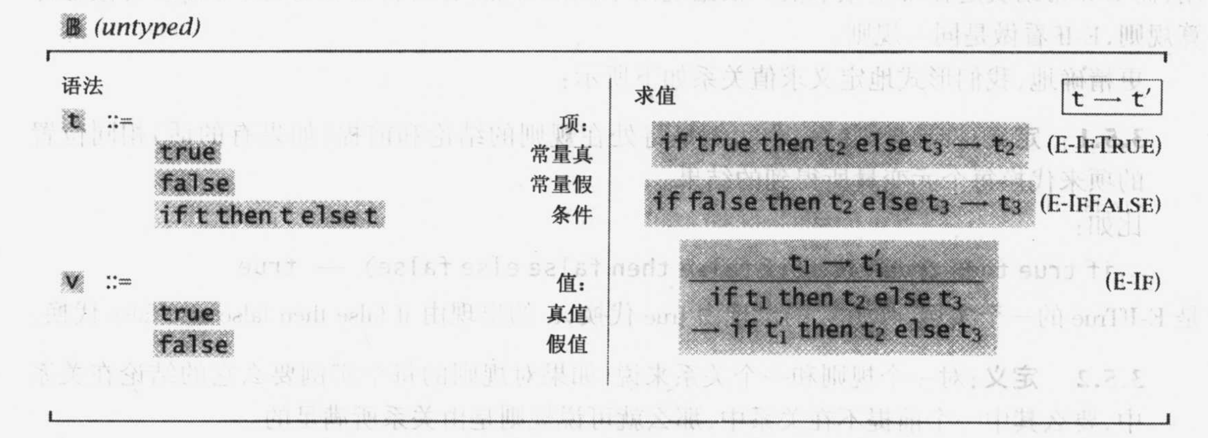


图 3.1 布尔值(B)

图 3.1 的左边一列是定义表达式的语法。第一个是项语法的重复表示(为了方便)。第二个定义项的子集，称为值，它可以是求值的最终结果。这里，值就是常量 true 和 false。本书中元变量 v 用于表示值。

右边一列定义项上的求值关系[①]，记为 $t \rightarrow t'$，读做"t 一步求值为 t′"。直观意思是如果 t 是抽象机器在某瞬间的状态，则机器能够做一步计算，并且将状态改变为 t′。这个关系是由 3 个推导规则所定义的(或者说是两个公理和一个规则，因为前两个没有前提)。

第一个规则 E-IfTrue 意思是：如果求值的项是一个条件句，它的条件是常量 true，则机器能丢掉条件表达式，将 then 部分 t_2 作为机器的新状态(即下一个要求值的项)。类似地，E-Iffalse 意思是条件为 false 的条件句在一步求值中求值为它的 else 分支 t_3。这些规则的名称中的 E 只是用于提醒人们是求值关系的一部分；其他关系的规则将会有不同的前缀。

第三个求值规则 E-If 更有趣。它说，如果前提 t_1 求值为 t'_1，则整个条件句：if t_1 then t_2 else t_3 求值为 if t'_1 then t_2 else t_3。借助于抽象机器，如果另一个机器的状态为 t_1，能一步到达状态 t'_1，则处于状态 if t_1 then t_2 else t_3 的机器能一步到达状态 if t'_1 then t_2 else t_3[②]。

有些虽然规则没有说明，但同样重要。常量 true 和 false 不会求值出什么结果，因为它们

① 有些专家喜欢采用术语"归约"表示这个关系，而将求值留在练习(3.5.17)中描述的"大步变式"，这个变式直接将项映射为最终值。

② 此处应为同一个机器——译者注。

不会出现在任何规则的左端。此外,没有规则允许在求值 if 本身之前先求值 if 的 then 或 else 子表达式:例如,项:

```
if true then (if false then false else false) else true
```

不能求值为 if true then false else true。我们惟一的选择是用 E-If 首先求值外边的条件句。与通常程序设计语言求值顺序的问题类似,这里将几个规则集合考虑进来会得出条件句特殊的求值策略:为求值一个条件句,先对它的条件求值;如果条件本身是一个条件句,必须求值这个条件的条件句,然后继续下去。E-IfTrue 和 E-IfFalse 规则告诉我们,当处理到最后发现该条件句的条件已经被求值后应该做什么。在某种意义下,E-IfTrue 和 E-IfFalse 确实起到了求值的作用,而 E-If 帮助决定在哪里做求值。根据规则不同的特点,有时将 E-IfTrue, E-IfFalse 看做是计算规则,E-If 看做是同一规则。

更精确地,我们形式地定义求值关系如下所示:

3.5.1 定义: 推导规则的一个实例是将处在规则的结论和前提(如果有的话)相同位置的项来代换每个元变量所得到的结果。

比如:

```
if true then true else (if false then false else false) ⟶ true
```

是 E-IfTrue 的一个实例,其中 t_2 的出现由 true 代换,t_3 的出现由 if false then false else false 代换。

3.5.2 定义: 对一个规则和一个关系来说,如果对规则的每个实例要么它的结论在关系中,要么其中一个前提不在关系中,那么就可说规则是由关系所满足的。

3.5.3 定义: 一步求值关系"⟶"是满足图 3.1 中 3 个规则的项上最小的二元关系。当序对(t, t′)在求值关系中,我们称求值语句(或判断)$t \longrightarrow t'$是可推导出的。

这里,"最小"是指一个语句 $t \longrightarrow t'$是可推导的,当且仅当它可由规则判断:要么它是公理 E-IfTrue 或E-IfFalse 的一个实例,要么它是前提为可推导规则 E-If 的一个实例的结论。一个给定语句的可推导性能通过推导树得出,这里推导树的叶子标记为 E-IfTrue 或 E-IfFalse 的实例,它的内部节点标记为 E-If 的实例。比如,为在本页写下推导,我们简记为:

```
s ≝ if true then false else false
t ≝ if s then true else true
u ≝ if false then true else true
```

则语句:

```
if t then false else false ⟶ if u then false else false
```

的可推导性由下列推导树验证:

$$\cfrac{\cfrac{\cfrac{}{\texttt{s} \longrightarrow \texttt{false}}\ \text{E-IfTrue}}{\texttt{t} \longrightarrow \texttt{u}}\ \text{E-If}}{\texttt{if t then false else false} \longrightarrow \texttt{if u then false else false}}\ \text{E-If}$$

称这个结构为树有点奇怪,因为它不含任何分叉。的确,验证求值语句的推导树总是有这种"简单"的形式:因为没有求值规则有多于一个的前提,所以没有分叉的推导树。推导树更适合讨论其他归纳定义的关系,如类型,它的某些规则有多个前提。

当推理求值关系的性质时，这个事实常常是有用的：一个求值语句 $t \to t'$ 是可推导的，当且仅当存在一个根标记为 $t \to t'$ 的推导树。特别是，这个事实直接引出一个称为对推导做归纳的证明技术。下面定理的证明说明了这个技术。

3.5.4　定理[一步求值的确定性]：如果 $t \to t'$ 且 $t \to t''$，则 $t' = t''$。

证明：对 $t \to t'$ 的一个推导做归纳。在归纳的每一步，假设对所有较小的推导结论成立，然后根据在推导的根部所用的求值规则进行分析（注意：这里的归纳不是对求值序列的长度做归纳，而只考虑一步求值。可以说是对 t 的结构做归纳，因为一个"求值推导"的结构直接来自于所约简项的结构。换言之，我们也能对 $t \to t''$ 的推导做归纳）。

如果在 $t \to t'$ 的推导中用到的最后的规则是 E-IfTrue，则知道 t 有形式：if t_1 then t_2 else t_3，其中 t_1 = true。但很明显 $t \to t''$ 的推导中的最后规则不可能是 E-IfFalse，因为我们不可能同时有 t_1 = true 和 t_1 = false。此外，在第二个推导中的最后规则也不可能是 E-If，因为这个规则的前提要求对某个 t_1'，有 $t_1 \to t_1'$，但我们已经注意到 true 不能求值到任何东西。因此，在第二个推导中的最后规则只能是 E-IfTrue，且直接有 $t' = t''$。

类似地，如果 $t \to t'$ 的推导中所用的最后规则是 E-IfFalse，则 $t \to t''$ 的推导中的最后规则必定是相同的，且直接得出结果。

最后，如果在 $t \to t'$ 的推导中所用的最后规则是 E-If，则这个规则的形式告诉我们 t 有形式 if t_1 then t_2 else t_3，其中对某个项 t_1'，有 $t_1 \to t_1'$。由上述的讨论，$t \to t''$ 的推导中的最后规则只能是 E-If，因此，t 有形式 if t_1 then t_2 else t_3（我们早已经知道的）并且对某个 t_1''，有 $t_1 \to t_1''$。但应用归纳假设（因为 $t_1 \to t_1'$ 和 $t_1 \to t_1''$ 的推导是原来 $t \to t'$ 和 $t \to t''$ 推导的子推导），得到 $t_1' = t_1''$。因此，t' = if t_1' then t_2 else t_3 = if t_1'' then t_2 else t_3 = t''。

3.5.5　练习[★]：用定理(3.3.4)的形式写出上一个证明用到的归纳原理。

这里的一步求值关系说明一个抽象机器在求值一个项时如何从一个状态移动到下一个状态。但作为程序员，只关心求值的最后结果，即机器不能再往下移动的状态。

3.5.6　定义：如果没有求值规则可以作用于项 t，则该项是范式，即不存在 t' 使得 $t \to t'$（有时说"t 是一个范式"，表示"t 是一个成范式的项"）。

我们已经注意到了 true 和 false 是当前系统中的范式（因为所有的求值规则的左端最外构造子是 if，明显地没有办法对任何规则实例化使得它的左端变成 true 或 false）。我们会用更一般术语重述这个关于值的事实。

3.5.7　定理：每个值都是范式。

当在系统中加入算术表达式（在以后章节中，还将加入其他构造），我们总能使定理(3.5.7)成立：值（即一个完全求值后的结果）的特点之一就是它为一个范式，任何语言的定义如果没有这个结论，则定义就不成立。

在当前的系统中，定理(3.5.7)的逆也是成立的：每个范式都是一个值，但推广开来，情况就不是这样；事实上，在后面将讨论算术表达式时，不是值的范式在运行时间错误分析中起着极其重要的作用。

3.5.8 定理:如果 t 是范式,则 t 是一个值。

证明:假设 t 不是一个值。由对 t 做结构归纳,很容易证明 t 不是一个范式。

由于 t 不是一个值,它必定有形式:对某些项 t_1,t_2 和 t_3,if t_1 then t_2 else t_3。考虑 t_1 的可能形式。

如果 t_1 = true 则显然 t 不是一个范式,因为它与 E-IfTrue 的左端匹配。类似地,考虑情况 t_1 = false。

如果 t_1 既不是 true 也不是 false,则它不是一个值。运用归纳假设,得到 t_1 不是一个范式,即存在某个 t_1' 使得 $t_1 \rightarrow t_1'$。但这意味着我们能用 E-If 推导 $t \rightarrow$ if t_1' then t_2 else t_3,所以t 也不是一个范式。

有时能将多步求值看做一大步状态转换是很方便的。我们定义一个多步求值关系,将一个项与所有能由它通过零步或几步求值推导出的项建立关系。

3.5.9 定义:多步求值关系"$\rightarrow^*$"是一步求值关系的自反,传递闭包。即它是最小的关系,使得(1)如果 $t \rightarrow t'$,则 $t \rightarrow^* t'$;(2)对所有的 t,有 $t \rightarrow^* t$;且(3)如果 $t \rightarrow^* t'$且 $t' \rightarrow^* t''$ 则 $t \rightarrow^* t''$。

3.5.10 练习[★]:将定义(3.5.9)表达为推导规则集。

有了多步求值这个概念以后,就很容易陈述下面的定理:

3.5.11 定理[范式的惟一性]:如果 $t \rightarrow^* u$ 且 $t \rightarrow^* u'$,其中 u 和 u'是范式,则u = u'。

证明:根据一步求值的确定性[定理(3.5.4)]进行推论可证。

在考虑算术表达式之前我们所要考虑关于求值的最后一个性质是:每个项能被求值到一个值。显然,这是另一个性质,但在更丰富的语言特征,如递归函数定义中不一定成立。即使在它成立的情况下,它的证明比我们将看到的要更加复杂。在第 12 章中,将回到这一点,说明一个类型系统如何为某种语言的一个可终止证明起关键作用。

计算机科学中大部分停机证明有相同的基本形式[①]:首先,选择某个良定集合 S 和一个将机器状态(这里指项)映射到 S 的函数f。然后,证明任何时候只要一个机器状态 t 能一步到达另一个状态 t',就有$f(t') < f(t)$。注意到从 t 开始的求值步骤的无穷序列通过f能映射到S 元素的无限递减序列。由于 S 是良定的,不存在这样的无限递减链,因此没有无限求值序列。函数f常常称为求值关系的一个"终止度量函数"。

3.5.12 定理[求值终止性]:对每个项 t,存在某个范式 t'使 $t \rightarrow^* t'$。

证明:注意到每求值一步,项的长度都在减少,因为项的长度(自然数的序是良定的)是个终止度量,所以求值结果为某个范式时,求值就终止了。

3.5.13 练习[推荐,★★]:

1. 假定在图 3.1 中加入一个新规则:

$$\texttt{if true then } t_2 \texttt{ else } t_3 \longrightarrow t_3 \qquad \text{(E-Funny1)}$$

---

① 在第 12 章中,将看到一个更复杂结构的停机证明。

上述定理[定理(3.5.4),定理(3.5.7),定理(3.5.8),定理(3.5.11)和定理 3.5.12)]中哪些仍成立?

2. 假定在图 3.1 中加这个规则:

$$\frac{\mathtt{t_2} \longrightarrow \mathtt{t_2'}}{\mathtt{if\ t_1\ then\ t_2\ else\ t_3} \longrightarrow \mathtt{if\ t_1\ then\ t_2'\ else\ t_3}} \quad \text{(E-Funny2)}$$

上面定理中哪些仍成立?这些证明需要改变吗?

下面我们将求值的定义推广到算术表达式。图 3.2 给出定义中新的部分(图 3.2 右上角的记法提醒我们将这个图看做是图 3.1 的扩展,而不是自身独立的语言)。

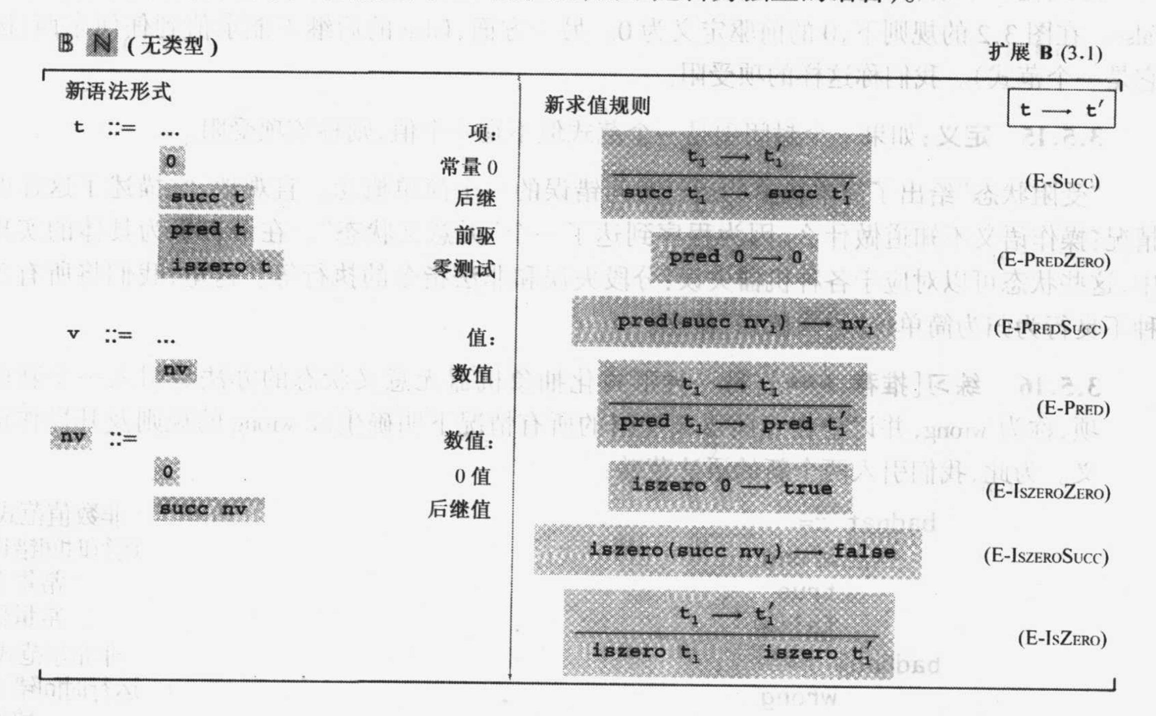


图 3.2　算术表达式(NB)

可以看出项的定义就是 3.1 节中语法的重复。值的定义有点意思,因为它引入了一个新的语法范畴——数值。直观理解是求值一个算术表达式的最后结果可以是一个数,这个数要么是0,要么是另一个数的后继[但不是任何值的后继:我们认为 succ(true)是一个错误,不是一个值]。

图 3.2 的右边一列中的求值规则与图 3.1 中形式相同。这里有 4 个计算规则(E-PredZero, E-PredSucc, E-IszeroZero 和 E-IszeroSucc)说明算子 pred, iszero 如何运用于数,且 3 个同一规则(E-Succ, E-Pred 和E-Iszero)将求值转为复合项的“第一个”子项。

严格地讲,我们应当重复定义(3.5.3)(“算术表达式的一步求值关系是满足图 3.1 和图 3.2 中规则的所有实例的最小关系”)。为避免浪费空间在重复上,通常将推导规则作为关系来定义,默认是“包含所有实例的最小关系”。

数值(nv)的语法范畴在这些规则中起着重要的作用。比如,在 E-PredSucc 中,左端是 pred(succ nv_1)[而不是 pred(succ t_1)]意味着这个规则不能用于求值 pred(succ(pred 0))到 pred(0),因为这将要求用 pred 0 实例化元变量 nv_1,但 pred 0 不是一个数值。项 pred(succ(pred 0))的求

值中惟一的下一步以下列推导树形式表示：

$$\dfrac{\dfrac{\dfrac{}{\texttt{pred 0} \longrightarrow \texttt{0}}\ \textsc{E-PredZero}}{\texttt{succ (pred 0)} \longrightarrow \texttt{succ 0}}\ \textsc{E-Succ}}{\texttt{pred (succ (pred 0))} \longrightarrow \texttt{pred (succ 0)}}\ \textsc{E-Pred}$$

3.5.14 练习[★★]：证明定理(3.5.4)对算术表达式的求值关系也成立：如果 $t \to t'$ 且 $t \to t''$，则 $t' = t''$。

形式化一个语言的操作语义，需要说明所有项的行为，包括刚用到的项，如 pred 0 和 succ false。在图 3.2 的规则下，0 的前驱定义为 0。另一方面，false 的后继不能求值到任何东西(因它是一个范式)。我们称这样的项受阻。

3.5.15 定义：如果一个封闭项是一个范式但不是一个值，则称该项受阻。

"受阻状态"给出了简单机器的运行时间错误的一个简单概念。直观地，它描述了这样的情况：操作语义不知道做什么，因为程序到达了一个"无意义状态"。在语言更为具体的实现中，这些状态可以对应于各种机器失误：分段失误和非法指令的执行等。这里，我们将所有各种不良行为归为简单的"受阻状态"概念。

3.5.16 练习[推荐，★★★]：另一种形式化抽象机器无意义状态的办法是引入一个新的项，称为 wrong，并讨论在当前语义受阻的所有情况下明确生成 wrong 的规则及其操作语义。为此，我们引入两个新的语法范畴：

badnat ::=	非数值范式：
wrong	运行时间错误
true	常量真
false	常量假
badbool ::=	非布尔范式：
wrong	运行时间错误
nv	数值

以及含下列规则的求值关系：

if badbool then t_1 else t_2 ⟶ wrong	(E-IF-WRONG)
succ badnat ⟶ wrong	(E-SUCC-WRONG)
pred badnat ⟶ wrong	(E-PRED-WRONG)
iszero badnat ⟶ wrong	(E-ISZERO-WRONG)

证明两个执行时间错误的处理是相同的：(1)找一个精确的方法陈述"两个处理是相同的"；(2)证明它。通常在证明关于程序语言的性质时，困难一点的是对要证明的精确语言进行形式化，而证明本身直接可得到。

3.5.17 练习[推荐，★★★]：操作语义的两个形式是通用的。在本书中采用的称为小步形式，因为求值关系的定义说明计算的各个步骤如何用于一步一步地重写项，直到它最终变成一个值。然后，定义一个多步求值关系，这允许我们讨论可以(多步)求值到值的项。另一种形式，称为大步语义(或有时称为自然语义)，直接形式化概念"该项求值为最终的

值”,记为$t \Downarrow v$。布尔和算术表达式语言的大步求值规则为:

$$v \Downarrow v \qquad \text{(B-VALUE)}$$

$$\frac{t_1 \Downarrow \texttt{true} \qquad t_2 \Downarrow v_2}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \Downarrow v_2} \qquad \text{(B-IFTRUE)}$$

$$\frac{t_1 \Downarrow \texttt{false} \qquad t_3 \Downarrow v_3}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \Downarrow v_3} \qquad \text{(B-IFFALSE)}$$

$$\frac{t_1 \Downarrow nv_1}{\texttt{succ } t_1 \Downarrow \texttt{succ } nv_1} \qquad \text{(B-SUCC)}$$

$$\frac{t_1 \Downarrow 0}{\texttt{pred } t_1 \Downarrow 0} \qquad \text{(B-PREDZERO)}$$

$$\frac{t_1 \Downarrow \texttt{succ } nv_1}{\texttt{pred } t_1 \Downarrow nv_1} \qquad \text{(B-PREDSUCC)}$$

$$\frac{t_1 \Downarrow 0}{\texttt{iszero } t_1 \Downarrow \texttt{true}} \qquad \text{(B-ISZEROZERO)}$$

$$\frac{t_1 \Downarrow \texttt{succ } nv_1}{\texttt{iszero } t_1 \Downarrow \texttt{false}} \qquad \text{(B-ISZEROSUCC)}$$

证明这个语言的小步和大步语义是一致的,即$t \longrightarrow^* v$,当且仅当$t \Downarrow v$。

3.5.18　练习[★★ ↛]:假设我们要改变语言的求值策略使得一个 if 表达式的 then 和 else 分支在条件求值之前被求值。说明如何改变求值规则来做到这一点。

3.6　注释

抽象和具体语法、分析等想法在许多关于编译的书籍都有定义。归纳定义,推导规则系统和归纳证明参见 Winskel(1993)和 Hennessy(1990)。

我们这里所用的操作语义的形式来自 Plotkin(1981)的技术报告。大步形式[练习(3.5.17)]来自 Kahn(1987)。进一步具体研究参见Astesiano(1991)和 Hennessy(1990)。

结构归纳是由 Burstall(1969)引入到计算机科学中来的。

> 问:为什么要对程序语言进行证明? 如果定义是正确的,这些证明总是令人厌烦的。
> 答:定义几乎总是错误的。
>
> ——佚名

第 4 章　算术表达式的一个 ML 实现[①]

形式定义，如前一章采用的形式，若能用一个具体的实现作为它成立的基础，则使用起来更容易。我们这里描述实现布尔和算术表达式的语言的关键部分（不想了解后面章节描述的类型检查器实现方式的读者可以跳过这一章及后面所有关于 ML 实现的章节）。

这里给出（和本书关于实现的章节中）的原码是用 ML 系列（Gordon，Milner 和 Wadsworth，1979）中的语言编写的，这个语言称为 Objective Caml（OCaml）（Leroy，2000；Cousineau 和 Mauny，1998）。这里，只用到了完整语言的一小子集；可以很容易地将例子转换成大部分其他语言。最主要的要求是自动存储管理（垃圾收集）和通过在结构数据类型上的模式匹配来定义递归函数的简易工具。其他函数式语言，像 Standard ML（Milner，Tofte，Harper 和 MacQueen，1997），Haskell（Hudak 等，1992；Thompson，1999），Scheme（Kelsey，Clinger 和 Rees，1998；Dybvig，1996）（含某模式匹配扩充形式）均可用来实现本章的内容。带有垃圾收集但没有模式匹配的语言，像 Java（Arnold 和 Gosling，1996）和纯 Scheme，对我们要做的程序有点困难。既没有垃圾收集又没有模式匹配的语言，像 C 语言（Kernighan 和 Ritchie，1988），就更不合适了[②]。

4.1　语法

我们的第一件事是定义用来表示项的 OCaml 值的类型。OCaml 数据类型定义机制很容易做到这一点：下面的说明就是对 3.1 节的语法的另一种表示方式。

```
type term =
    TmTrue of info
  | TmFalse of info
  | TmIf of info * term * term * term
  | TmZero of info
  | TmSucc of info * term
  | TmPred of info * term
  | TmIsZero of info * term
```

构造子从 TmTrue 到 TmIsZero 都是给类型 term 的抽象语法树上不同种类的节点命名；在每种情况下 of 后面的类型说明该节点类型上子树的个数。

每个抽象语法树节点标记一个类型为 info 的值，它描述节点起源于哪里（源文件所在的位置）。

这个信息是由语法分析器在扫描输入文件时产生的，打印函数用它指明用户错误出现在哪里。为了理解求值、类型检查等基本算法，这个信息也是可以忽略的；它写在这里是为了让希望自己实现的读者可以看见与本书中讨论的形式相同的原代码。

---

① 本章中的代码可以在 http://www.cis.upenn.edu/~bcpierce/tapl 的 arith 实现中找到，可以按提示下载和建立这些实现。

② 当然，语言的口味是变化的，并且好的程序员会用任何可以使用的工具来完成任务；读者可以自由地使用任何喜欢的语言。但注意：一个类型检查器需要处理各种符号，为此手工完成存储管理是一件繁琐和易出错的事情。

在求值关系的定义中，我们需要检查一个项是否是一个数值：

```
let rec isnumericval t = match t with
    TmZero(_) → true
  | TmSucc(_,t1) → isnumericval t1
  | _ → false
```

这是一个 OCaml 中由模式匹配来递归定义的典型例子：isnumericval 定义为函数，当应用于 TmZero 时，输出 true，而应用于带子树 t1 的 TmSucc 时，将做一个递归调用来检查 t1 是否是一个数值；并且当应用于任何其他项时，输出 false。下划线(_)表示任意与此处的项匹配的内容；它们用在头两个子句中只是为了避开遇到 info 注释，用于最后的子句用来匹配任何项。rec 关键字告诉编译器这是一个递归函数定义，即在它的体中对 isnumericval 的引用是指向正在定义的函数，而不是某个以前的有着相同名称的绑定。

注意上面定义中的 ML 原代码，为了易读性并保持与 lambda 演算的例子一致，在排版时可以有几种方法来美化。比如，用一个实箭头符号(→)，而不用两个字符(->)的序列。这些美化后的符号完整列表可以在本书的网站中找到。

检查项是否为值的函数类似：

```
let rec isval t = match t with
    TmTrue(_)  → true
  | TmFalse(_) → true
  | t when isnumericval t  → true
  | _ → false
```

第 3 个子句是一个“条件模式”：，只有布尔表达式 isnumericval t 结果为 true，它才与任何项匹配。

4.2　求值

求值可以完全按照图 3.1 和图 3.2 中单步求值规则来实现。如我们所见，这些规则定义一个部分函数，当将它们应用于还不是值的项时，产生这个项的下一步求值。当应用于一个值时，求值函数不产生任何结果。为了将求值规则转换为 OCaml，需要找出相应的解决办法。一个直接的办法是写单步求值函数 eval1，当没有求值规则可应用于所给的项时它提升一个异常(另一个可能的办法是使单步求值送回一个 term option，指明求值是否成功了，如果成功，给出结果项；这种方法也能成功实现，只是需要多一点信息文档)。我们首先定义当没有求值规则可运用时会出现的异常：

```
exception NoRuleApplies
```

现在可以写出单步求值器了：

```
let rec eval1 t = match t with
    TmIf(_,TmTrue(_),t2,t3) →
      t2
  | TmIf(_,TmFalse(_),t2,t3) →
      t3
  | TmIf(fi,t1,t2,t3) →
```

```
        let t1' = eval1 t1 in
        TmIf(fi, t1', t2, t3)
    | TmSucc(fi,t1) →
        let t1' = eval1 t1 in
        TmSucc(fi, t1')
    | TmPred(_,TmZero(_)) →
        TmZero(dummyinfo)
    | TmPred(_,TmSucc(_,nv1)) when (isnumericval nv1) →
        nv1
    | TmPred(fi,t1) →
        let t1' = eval1 t1 in
        TmPred(fi, t1')
    | TmIsZero(_,TmZero(_)) →
        TmTrue(dummyinfo)
    | TmIsZero(_,TmSucc(_,nv1)) when (isnumericval nv1) →
        TmFalse(dummyinfo)
    | TmIsZero(fi,t1) →
        let t1' = eval1 t1 in
        TmIsZero(fi, t1')
    | _ →
        raise NoRuleApplies
```

注意有几处的项是随意构造的，并不是将已存在的项重新组合。因为这些新项在用户原来的源文件中并不存在，它们的 info 注释没有用处。在这些项中常量 dummyinfo 起到了 info 注释的作用。文件信息的变量名称 fi 用于在模式中匹配 info 注释。

在 eval1 的定义中，另一点需要注意的是在模式中使用 when 子句来捕捉元变量名称，如 v 和 nv 在图 3.1 和图 3.2 求值关系表示中的效果。比如，在求值 TmPred(_,TmSucc(_,nv1))的子句中，OCaml 模式的语义允许 nv1 去匹配任何不是我们所要的项；用 when(isnumericval nv1)来限制规则，这样只有当由 nv1 匹配的项实际是一个数值时才调用这个规则(如果需要，我们可以用与 ML 模式相同的形式来重写原来的推导规则，将元变量名称中的隐含约束转为规则上明显的边界条件：

$$\frac{t_1\text{ 是一个数值}}{\texttt{pred (succ } t_1\texttt{)} \longrightarrow t_1} \qquad \text{(E-PredSucc)}$$

但这样做会影响紧致性和可读性。

最后，eval 函数取一个项，通过重复调用 eval1 找到它的范式。只要 eval1 返回一个新项 t′，就继续对 eval 递归调用，求值 t′。当 eval1 最后到达没有规则可运用时，它提升异常 NoRuleApplies，造成 eval 中断循环，并返回序列中的最后一个项[①]：

```
let rec eval t =
  try let t' = eval1 t
      in eval t'
  with NoRuleApplies → t
```

4.2.1 练习[★★]：什么才是 eval 更好的写法？

① 我们这样使用 eval 是为了简化，但将一个 try 句柄放在一个递归回路中在 ML 中不是非常好的形式。

显然，这个简单求值器是为了与求值的数学定义相比较，而不是尽快地找到范式。一个更有效的算法可以用练习(4.2.2)中的大步求值规则。

4.2.2　练习：[推荐，★★★ ↛]：在 arith 的实现中改变函数 eval 的定义为练习(3.5.17)中引入的大步形式。

4.3　其余部分

当然，除了我们这里讨论的之外，解释器或编译器还有许多其他部分——即使这些部分非常简单。实际上，求值的项在文件中开始时是一个字符的序列。在我们已知的函数实际求值它们之前，必须从文件系统中读出，由一个词法分析器生成符号流，然后由一个语法分析器生成抽象语法树。此外，在求值之后，结果需要打印出来：

[文件 I/O] —字母→ [词法分析] —符号→ [语法分析] —项→ [求值] —值→ [打印]

感兴趣的读者可以看一下在线解释器的 OCaml 原代码。

第5章　无类型 lambda 演算[①]

这一章我们将回顾无类型或纯 lambda 演算的定义和基本性质，这是本书将描述的大部分类型系统的“计算基础”。

在 20 世纪 60 年代中期，Peter Landin 注意到一个复杂程序语言可以这样来理解：将它形式化为反映语言基本机制的微小核心演算，加上一个便利的推导形式的集合，其推导行为可以转换到核心演算中（参见 Landin 1964，1965，1966；也可参见 Tennent 1981）。Landin 用的核心语言是 lambda 演算，由 Alonzo Church（1936，1941）于 20 世纪 20 年代发明的一个形式系统，其中所有的计算归约为函数定义和应用的基本运算[②]。由于 Landin 的见解和 John McCarthy 于 20 世纪 70 年代在 LISP（1959，1981）方面的开创性工作，lambda 演算广泛用于程序语言特征的说明、语言设计和实现，以及类型系统的研究。它的重要性在于它既可作为描述计算的一个简单程序语言，同时也可作为一个数学对象，其中的严格语句能得到证明。

lambda 演算只是有着类似目的的许多核心演算之一。Milner，Parrow 和 Walker（1992，1991）的 pi 演算已经成为定义基于消息并发语言的语义核心语言，而 Abadi 和 Cardelli 的对象演算（1996）精炼出面向对象语言的核心特征。我们讨论 lambda 演算中涉及到的大部分概念和技术都能直接转换到其他演算中去。第 19 章将讨论这种转换的一个实例。

lambda 演算可以从不同的方面加以丰富。首先，常常容易加入那些行为在核心语言中已经模拟的特征。如数、元组、记录等的具体语法。更有趣的是，我们还能加入更复杂特征，像易变引用单元或非局部异常处理，它们在核心语言中只有经过复杂的翻译才能建模。这样的扩展最终产生了如 ML（Gordon，Milner，Wadsworth 1979；Milner，Tofte 和 Harper，1990；Weis，Aponte，Laville，Mauny 和 Suárez，1989；Milner，Tofte，Harper 和 MacQueen，1997），Haskell（Hudak 等，1992），或 Scheme（Sussman 和 Steele，1975；Kelsey，Clinger 和 Rees，1998）这样的语言。我们在以后章节中将看到对核心语言的扩展常常包含对类型系统的扩展。

5.1　基础

过程（或函数）抽象基本上是所有程序语言的关键特征。不用重复写一个演算，我们借助于一个或多个命名参数写一个函数或过程的一般执行演算，并且需要时实例化这个函数，在每个情况下提供参数的值。比如，一个程序员需要耐心重复地写这样的表达式，如：

```
(5*4*3*2*1) + (7*6*5*4*3*2*1) - (3*2*1)
```

并重写为 factorial(5) + factorial(7) - factorial(3)，其中：

```
factorial(n)  =  if n=0 then 1 else n * factorial(n-1)
```

---

① 本章中的例子是根据纯无类型 lambda 演算，λ（参见图 5.3）或扩展为带布尔和算术操作的 lambda 演算，λ **NB**（3.2）。相关的 OCaml 实现是 fulluntyped。

② 此处应为函数抽象——译者注。

对每个非负数 n，用参数 n 实例化函数 factorial 产生 n 的阶乘。如果用"λ n…"表示"函数对每个 n，产生……"我们能将 factorial 的定义写为：

```
factorial  =  λn. if n=0 then 1 else n * factorial(n-1)
```

则 factorial(0)意思是"函数(λn. if n = 0 then 1 else…)应用于参数 0"，即变量 n 在函数体(λn. if n = 0 then 1 else…)中用 0 代换的结果值，即，"if 0 = 0 then 1 else…，"值为 1。

lambda 演算(或 λ 演算)用最可能纯的形式表示这种函数定义和应用。在 lambda 演算中，每个事物均是一个函数：函数接受的参数是函数，一个函数的结果是另一个函数。

lambda 演算的语法由 3 种项组成[①]。一个变量 x 本身是一个项；一个变量 x 在项 t_1 中的抽象，记为 $\lambda x.t_1$，是一个项；并且一个项 t_1 应用于另一个项 t_2 时，记为 $t_1\ t_2$，也是一个项。形成项的这些方式可用下列语法表示：

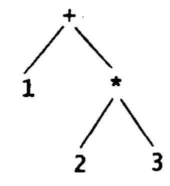
```
t ::=                                   项：
      x                                 变量
      λx.t                              抽象
      t t                               应用
```

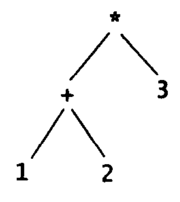
下面的小节将详细地讨论这个定义。

抽象语法和具体语法

当讨论程序语言的语法时，要区别结构的两个层次[②]。语言的具体语法(或表面语法)指程序员直接读和写的字符串。抽象语法是将程序表示为标签树的更为简单的内部表示(称为抽象语法树或 AST)。树表示使项的结构明显化，无论是在严格的语言定义(和它们的证明)中还是编译器和解释器的内部，都适合进行复杂处理。

从具体语法到抽象语法的转换由两步组成。首先，一个词法分析器(或词法器)将程序员写的字符串转换为符号序列——标识符、关键字、常量和标点等。词法器除掉注解，同时处理如空格、大小写约定，以及数和串常量的格式等问题。接着，一个语法分析器将标记序列转换为一个抽象语法树。在分析时，各种约定，如算子的优先权和结合律，可以减少程序中表明复合表达式层次结构的括号数量。比如，* 优先于 +，因此语法分析器解释无括号的表达式 1 + 2 * 3 为一个左边的抽象语法树。





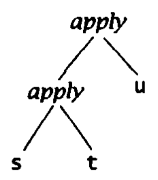
本书的重点是抽象语法，不是具体语法。像上面的 lambda 项的语法应理解为描述合法的树结构，而不是符号或字符的串。当然，当我们写例子、定义、定理和证明中的项时，需要将它们表示为一个具体的线性符号，但我们要记住它们所表达的是抽象语法树。

---

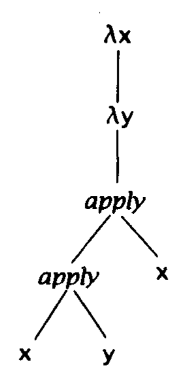
① 术语 lambda 项是指 lambda 演算中的任意项。以 λ 开头的 lambda 项常常称为 lambda 抽象。

② 成熟语言的定义有时采用多层。比如，根据 Landin，将某些语言构造的行为定义(通过将它们转换为其他更基本特征的组合定义)为导出形式，常常是非常有用的。只包含核心特征的子语言称为内部语言(IL)，而包含所有导出形式的完全语言称为外部语言(EL)。经过分析，从 EL 到 IL 的转换(至少概念上)将独立进行。在 11.3 节中将讨论导出形式。

为省略太多的括号，当我们以线性方式写 lambda 项时将采用两个约定。首先，应用的是左结合，即 s t u 和(s t) u 表示相同的树。



其次，抽象体尽可能右扩展，使得比如，λx.λy.x y x 和 λx.(λy.((x y)x))表示相同的树。



变量和元变量

上面讨论的语法定义中另一难点是使用元变量。我们将继续用元变量 t(以及 s 和 u，带下标或不带下标)表示任意项[①]。类似地，x(以及 y 和 z)表示一个任意变量。注意，这里的 x 是一个在变量上取值的元变量。由于短名称集合有限，我们仍用 x，y，…表示对象语言中的变量。在这种情况下，只要根据上下文就能知道谁是变量，谁是元变量。比如，在句子"项 λx.λ*y*.*xy* 有形式 λz.s，其中 z = x 且 s = λy.xy"中，名称 z，s 是元变量，而 x 和 y 是对象语言变量。

辖域

关于 lambda 演算的语法我们将讨论的最后一点是变量的辖域。

对变量 x，当它出现在抽象 λx.t 的 t 中时，则认为 x 的出现是囿界(或称被界定)的(更精确地说，它被这个抽象所界定。等价地，也可以说 λx 是一个绑定器，它的辖域是 t)。x 是自由的，如果它出现的位置不被任何对 x 的抽象所界定。比如，x 在 xy 和 λy.xy 中是自由的，而在 λx.x 和 λz.λx.λy.x(yz)中的出现是囿界的。在(λx.x)x 中的第一个出现是囿界的，而第二个则是自由的。

一个不含自由变量的项称为封闭项；封闭项也称为组合子。最简单的组合子，称为恒等函数：

```
id = λx.x;
```

只输出它的变元。

① 自然地，在这一章中，t 表示 lambda 项，而不是在算术表达式。在本书中，t 总是表示于所讨论演算的项。每一章第一页的脚注都会说明这个系统是什么系统。

操作语义

纯形式(或纯)的 lambda 演算没有内置的常量或原语算子——没有数、算术运算、条件子、记录、循环、序列、I/O，等等。项计算的惟一含义是将函数应用到参数(参数本身是函数)。计算的每一步是重写一个左端部分为抽象的应用，用右端部分代换抽象体中的囿变量。图示记为：

$$(\lambda x.\ t_{12})\ t_2 \longrightarrow [x \mapsto t_2]t_{12},$$

其中$[x \mapsto t_2]t_{12}$表示“由 t_2 代换在 t_{12} 中所有自由出现的 x 得到的项”。比如，项(λx.x)y 求值为 y，项(λx.x(λx.x))(u r)求值为 u r(λx.x)。采用 Church 的概念，形为$(\lambda x.t_{12})t_2$ 的项称为一个约式(“可归约表达式”)，并且根据上述规则重写一个约式的操作称为 beta 归约。

程序语言设计者和理论工作者在多年的研究中提出了 lambda 演算中几个不同的求值策略。每个策略确定一个项在下一步求值中激活哪一个约式或几个约式[①]。

- 在全 beta 归约下，任何时刻可以归约任何一个约式。在每一步，我们对进行求值的项，取任何位置的某个约式，并且归约它。比如，考虑项：

  (λx.x) ((λx.x) (λz. (λx.x) z)),

  可以记为 id(id(λz.id z))。这个项包含 3 个约式：

  <u>id (id (λz. id z))</u>
  id <u>(id (λz. id z))</u>
  id (id (λz. <u>id z</u>))

  在全 beta 归约下，我们可以选择，比如，最里面的约式开始归约，然后对中间的约式，最后对最外面的约式进行归约：

  ```
     id (id (λz. id z))
  →  id (id (λz.z))
  →  id (λz.z)
  →  λz.z
  ↛
  ```

- 采用规范顺序策略，最左边，最外面的约式总是第一个被归约。在这种策略下，上面的项可以归约如下：

  ```
     id (id (λz. id z))
  →  id (λz. id z)
  →  λz. id z
  →  λz.z
  ↛
  ```

  在这种(以及下面的几种)策略下，求值关系实际上是一个部分函数：每个项 t 一步至多求值为一个项 t′。

- 按名调用策略限制条件更多，不允许在抽象内部进行归约。对上述项，我们可以先按照规范顺序策略做前两个归约，但在遇到最后一个时停止，将 λz.id z 看做是一个范式：

---

① 有些人不区分使用术语“归约”和“求值”。有时用“求值”只表示涉及某些“值”概念的策略，而用“归约”表示其他策略。

```
    id (id (λz. id z))
→   id (λz. id z)
→   λz. id z
↛
```

某些著名的程序语言，如 Algol 60(Naur 等，1963)和 Haskell(Hudak 等，1992)采用按名调用的变化形式。Haskell 实际上采用一个优化的形式，称为按需调用(call by need)(Wadsworth 1971；Ariola 等，1995)，不用每次重新求值所有用到的变元，而用它第一次求出的值覆盖变元的所有出现值，以避免以后重新求值。这个策略要求我们在项的执行时间表示中维护某个共享。事实上，这是一个抽象语法图上的归约关系，而不是语法树上的归约关系。

- 大部分语言采用按值调用(call by value)的策略，这里只有最外面的约式可以归约，并且只有当该约式的右边均已经归约到一个值时才能进行归约，这里的值是指这样一个项：它的计算已经完成，并且不能再归约了[①]。在这个策略下，上述例子的项归约如下：

```
    id (id (λz. id z))
→   id (λz. id z)
→   λz. id z
↛
```

  按值调用策略是严格的，意思是对函数的参数均求值，不管它们是否在函数体中发挥作用。相比之下，非严格的(或懒惰的)策略像按名调用以及按需调用只对实际要用到的参数求值。

当考虑类型系统时，选择哪种求值策略实际上没有多少差别。引发各种类型特征，以及说明它们所用的技术对所有的策略来说几乎是相同的。在本书中，我们采用按值调用，这是因为在大部分著名的语言中采用的都是这种策略，同时也因为它最容易用一些特征，如异常(参见第 14 章)和引用(参见第 13 章)来扩充。

5.2 lambda 演算中的程序设计

lambda 演算比其定义的形式更为强大。在本节中，我们将考虑几个用 lambda 演算表示程序的典型例子。这些例子并不是用来表明 lambda 演算应该看做是成熟的程序设计语言(所有广泛使用的高级语言提供更清晰和更有效的方法实现相同的任务)，而只是作为热身练习来感觉一下这个系统。

多参数

首先我们注意到 lambda 演算并没有对多参数函数提供内在支持。当然，加上这样的支持不是一件困难的事情，但更容易的办法是采用以函数为其结果的高阶函数来达到相同的效果。假定 s 是一个包含两个自由变量 x，y 的项，要写一个函数 f 使得对每个参数序对(v，w)产生在 s 中用 v 代换 x，并且用 w 代换 y 的结果。在一个较丰富的程序语言中，我们可以将 f = λ(x，y).s 写成 f = λx.λy.s。即 f 为一个函数，当用 v 代换 x 时会产生一个新的函数，再将新函数中的 y 用 w

① 在目前初级阶段的演算中，惟一的值是 lambda 抽象。在更高级的演算中将包含其他值：数值和布尔常量、串、值的元组、值的记录和值的列表等。

代换时就产生最终结果。然后我们一次一个运用 f 到它的参数，记为 f v w[即(f v)w]归约为((λy.[x ↦ v]s)w)，然后产生[y ↦ w][x ↦ v]s。这个多参数函数到高阶函数的转换称为 currying，以纪念 Haskell Curry，他是与 Church 同时代的人。

Church 布尔式

另一个可以很容易嵌入到 lambda 演算的语言特征是布尔值和条件式。定义项 tru 和 fls 如下所示：

```
tru = λt. λf. t;
fls = λt. λf. f;
```

(这些简写的名称是为了避免与第 3 章中原始布尔常量 true 和 false 相混淆)。

项 tru 和 fls 可以表示布尔值"真"和"假"，这样我们可以用这些项执行测试一个布尔真假值的操作。特别，可以用应用来定义一个组合式 test，使得当 b 是 tru 时，test b v w 归约为 v，而当 b 为 fls 时，test b v w 归约为 w：

```
test = λl. λm. λn. l m n;
```

组合式 test 实际上并没有什么用处：test b v w 可以归约为 b v w 。事实上，布尔子 b 本身是一个条件子：它取两个参数，然后选择第一个(如果它是 tru)或第二个(如果它是 fls)参数。比如，项 test tru v w 归约如下：

```
    test tru v w
=   (λl. λm. λn. l m n) tru v w    由定义
→   (λm. λn. tru m n) v w          归约下划线的约式
→   (λn. tru v n) w                归约下划线的约式
→   tru v w                        归约下划线的约式
=   (λt.λf.t) v w                  由定义
→   (λf. v) w                      归约下划线的约式
→   v                              归约下划线的约式
```

我们也可以像逻辑连接词一样定义布尔算子为函数：

```
and = λb. λc. b c fls;
```

即 and 是一个函数对给定两个布尔值 b 和 c，如果 b 是 tru，输出 c；并且如果 b 是 fls，输出 fls；这样，如果 b 和 c 两个均为 tru，and b c 产生 tru，如果要么 b，要么 c 是 fls 产生 fls。

```
  and tru tru;
▸ (λt. λf. t)

  and tru fls;
▸ (λt. λf. f)
```

5.2.1　练习[★]：定义逻辑 or 和 not 函数。

序对

利用布尔式，我们能定义值的序对为项：

```
pair = λf.λs.λb. b f s;
fst = λp. p tru;
snd = λp. p fls;
```

即 pair v w 是这样的一个函数,当应用于一个布尔值 b 时,b 会应用于 v 和 w。由布尔式的定义,如果 b 是 tru;这个应用产生 v;如果 b 是 fls 并且 w,因此,第一个和第二个投影函数 fst 和 snd 可以简单地用适当的布尔式来实现。为验证 fst(pair v w)→* v,可以计算如下:

```
     fst (pair v w)
=    fst ((λf. λs. λb. b f s) v w)    由定义
→    fst ((λs. λb. b v s) w)          归约下划线的约式
→    fst (λb. b v w)                  归约下划线的约式
=    (λp. p tru) (λb. b v w)          由定义
→    (λb. b v w) tru                  归约下划线的约式
→    tru v w                          归约下划线的约式
→*   v                                如前.
```

Church 数值

用 lambda 项表示数比我们在上面所看到的要复杂一些。定义 Church 数值 $c_0, c_1, c_2, \cdots$ 为:

```
c0 = λs. λz. z;
c1 = λs. λz. s z;
c2 = λs. λz. s (s z);
c3 = λs. λz. s (s (s z));
etc.
```

即每个数 n 用一个组合式 c_n 表示,这个组合式有两个参数,s 和 z(表示"后继"和"零"),并且应用 s 到 z,n 次。与布尔式和序对的情况一样,这种编码使数变为活跃的实体:数 n 用函数表示,该函数做某件事情 n 次(一种活跃的一元数字)。

(读者也许早已注意到了 c_0 和 fls 实际上是同一个项。在汇编语言中类似的"双关语"是普遍的,这里相同的比特模式可以表示许多不同的值,一个整数、一个浮点、一个地址、四个字符,等等,依赖于如何解释它;在 C 这样的低级语言中,0 和 false 也是相等的)。

我们可以定义 Church 数值上的后继函数为:

```
scc = λn. λs. λz. s (n s z);
```

项 scc 是这样的一个组合式,取一个 Church 数值 n,得出另一个 Church 数值,即它产生一个取参数 s 和 z,将 s 重复应用于 z 的函数。首先将 s 和 z 作为参数传递给 n 得到 s 应用于 z 的正确数值,然后再将 s 应用于这个结果。

5.2.2 练习[★★]:用另一种方法定义 Church 数值上的后继函数。

类似地,可以用项 plus 做 Church 数值的加法,这个项取两个 Church 数值 m,n 作为参数,产生另一个 Church 数值,即一个函数,它接受两个参数 s 和 z,重复应用 s 到 z,n 次(将 s,z 作为参数传给 n),然后应用 s 到这个结果 m 次:

```
plus = λm. λn. λs. λz. m s (n s z);
```

乘法的实现采用另一个技巧:因为 plus 一次取一个参数,应用它到一个参数 n 将产生这样的函数,将 n 与所给的任何参数相加。将这个函数作为第一个参数传给 m,c_0 作为第二个参数是指"将 n 加到函数的参数中,再将函数应用于 0,重复 m 次",即"将 n 叠加 m 次"。

```
times = λm. λn. m (plus n) c0;
```

5.2.3 练习[★★]:是否可能不用 plus 定义 Church 数值上的乘法?

5.2.4　练习[推荐,★★]:定义一个项将一个数变为另一个数的幂。

测试一个 Church 数值是否为零,我们必须找到某个合适的参数序对——特别是,必须应用数值到一个项 zz 和 ss 的序对使得应用 ss 到 zz 一次或多次产生 fls,而不是产生 tru。显然,应该取 zz 为 tru。对于 ss,可用一个不考虑参数且总是返回 fls 的函数:

```
iszro = λm. m (λx. fls) tru;

iszro c₁;
▸ (λt. λf. f)

iszro (times c₀ c₂);
▸ (λt. λf. t)
```

令人惊讶的是,Church 数值的减法比加法要困难一些。用下面的"前驱函数"能够定义数值的减法,这个"前驱函数"当给定 c_0 作为参数时,仅返回 c_0,当给定 c_{i+1} 作为参数时产生 c_i:

```
zz = pair c₀ c₀;
ss = λp. pair (snd p) (plus c₁ (snd p));
prd = λm. fst (m ss zz);
```

这个定义用 m 作为一个函数,将函数 ss 的 m 次拷贝应用到开始的值 zz 。每个 ss 的拷贝取数值序对 pair c_i c_j 作为它的参数,并产生 pair c_j c_{j+1} 作为其结果(如图 5.1 所示)。因此,运用 ss, m 次到 pair c_0 c_0,当 m = 0 时产生 pair c_0 c_0;并且当 m 为正时,产生 pair c_{m-1} c_m。在两种情况中,m 的前继出现在第一个部分中。

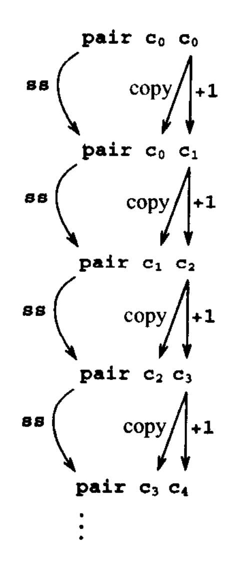


图 5.1　前驱函数的"内循环"

5.2.5　练习[★★]:用 prd 定义一个减法函数。

5.2.6　练习[★★]:估计一下计算 prd c_n 需要多少步求值(作为 n 的函数)?

5.2.7　练习[★★]:写出一个函数 equal 测试两个数是否相等,送回一个 Church 布尔式。比如:

```
  equal c₃ c₃;
▸ (λt. λf. t)

  equal c₃ c₂;
▸ (λt. λf. f)
```

其他常用的数据类型,像列表、树、数组和各种的记录能采用类似的技巧定义。

5.2.8 练习[推荐,★★★]: 一个列表能用函数 fold 在 lambda 演算中表示(这个函数的 OCaml 命名是 fold_left;它有时也称为 reduce)。比如,将列表[x,y,z]变成一个函数,它取两个参数 c 和 n,返回 cx(cy(czn)))。nil 该如何表示?写出一个函数 cons,它取一个元素 h 和一个列表 t(即一个 fold 函数)作为参数,返回一个将 h 插入到 t 的头部的新列表。写出 isnil 和 head 函数,每个取一个列表作为参数。最后,针对这个列表,写出一个函数 tail(这有点困难,需要利用类似于定义数的 prd 技巧)。

丰富演算

我们已经看到布尔式、数及其运算能在纯 lambda 演算中表示。的确,严格地讲,我们能在纯系统范围中进行任何程序设计。但是,当讨论例子时,常常也需要用上原始的布尔类型和数型(可能还有其他数据类型)。当我们需要明确是哪一个系统时,可用符号 λ 表示在图 5.3 中定义的纯 lambda 演算,λNB 表示含图 3.1 和图 3.2 中的布尔类型和算术表达式的系统。

在 λNB 中,当我们写程序时,实际上有两个不同的布尔类型和数的实现方法可以选择:一个是实际的实现,另一个就是本章中定义的代码。当然,很容易在两者之间进行转换。为将一个 Church 布尔类型转换为一个原始布尔类型,可将它应用到 true 和 false:

```
realbool = λb. b true false;
```

另一个方法是,用一个 if 表达式:

```
churchbool = λb. if b then tru else fls;
```

在高级操作中能够建立这些转换。这里,一个 Church 数值上的相等函数送回一个真实的布尔式:

```
realeq = λm. λn. (equal m n) true false;
```

用同样的方法,可以运用到 succ 和 0:

```
realnat = λm. m (λx. succ x) 0;
```

将一个 Church 数值转换到相应的原始数。不能直接运用 m 到 succ,因为 succ 本身没有语法意义:由我们定义算术表达式语法的方法,总能运用 succ 到某个项上。我们可以绕开这一点,将 succ 包装在一个小函数中,这个小函数不做其他事情,只送回 succ 的参数。

在例子中使用原始布尔式和数的理由主要与求值次序有关。比如,考虑项 scc c_1。从上面的讨论中,我们期望这个项应该求值为一个 Church 数值 c_2。事实上,它不是:

```
  scc c₁;
▸ (λs. λz. s ((λs'. λz'. s' z') s z))
```

这个项包含一个约式,如果我们要归约这个约式(在两步内)到达 c_2,但按值调用求值规则不允许我们归约它,因为它是一个 lambda 抽象。

这不是一个基本问题:由 scc c_1 的求值得到的项行为等价于 c_2,这是指运用它到任何参数

序对 v,w 产生结果与运用 c_2 到 v 和 w 的结果相同。剩余的计算使得验证函数 scc 的行为是否如我们期望的那样变得有点困难。对于更复杂的算术计算,情况将更糟。比如,times c_2 c_2 求值结果不是 c_4,而是下面的奇怪结果:

```
times c2 c2;
▸ (λs.
     λz.
       (λs'. λz'. s' (s' z')) s
       ((λs'.
     λz'.
       (λs". λz". s" (s" z")) s'
       ((λs". λz".z") s' z'))
   s
   z))
```

一种验证这个项的行为是否等同于 c_4 的方法是测试它们是否相等:

```
equal c4 (times c2 c2);
▸ (λt. λf. t)
```

但更直接的方法是取 times c_2 c_2,并将它转换为一个原始数:

```
realnat (times c2 c2);
▸ 4
```

这个转换有这样一个效果:提供 times c_2 c_2 所需的两个额外的参数,促使它体中所有潜在的计算发生。

递归

回忆一下,一个在求值关系下没有下一步的项称为一个范式。有趣的是,某些项不能求值到一个范式。比如,发散组合式:

```
omega = (λx. x x) (λx. x x);
```

只包含一个约式,并且归约这个约式将产生另一个 omega。没有范式的项称为发散的。

组合式 omega 有一个有用的推广,称为不动点组合式[①],这个组合式可以用来定义像 factorial 这样的递归函数[②]:

```
fix = λf. (λx. f (λy. x x y)) (λx. f (λy. x x y));
```

如同 omega,组合式 fix 有一个重复的结构;只根据它的定义很难理解这一点。最好的理解办法大概是看它在特殊的例子上如何运用的[③]。假定我们要写一个形为 h = 〈包含 h 的函数体〉的递归函数定义,即要写一个定义,其中 = 右边的项用到了我们正在定义的函数,如 5.1 节中 factorial 的定义。目的是递归定义应该在它出现的某一点上展开;比如,factorial 的定义直观上可以是:

---

① 这有时称为值调用 Y 组合式。Plotkin(1975)称它为 Z。

② 注意更简单的按名调用不动点组合式:

$$Y = \lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$$

在按值调用的形式中是无用的,因为对任何 g,表达式 Y g 发散。

③ 也可能从第一原则(Friedman 和 Felleisen,1996,第 9 章)中导出 fix 的定义,但这样的导出也是相当复杂的。

```
if n=0 then 1
else n * (if n-1=0 then 1
          else (n-1) * (if (n-2)=0 then 1
                        else (n-2) * ...))
```

或借助于 Church 数值：

```
if realeq n c0 then c1
else times n (if realeq (prd n) c0 then c1
              else times (prd n)
                         (if realeq (prd (prd n)) c0 then c1
                          else times (prd (prd n)) ...))
```

可以用组合式 fix，首先定义 g = λf.〈含 f 的函数体〉，然后定义 h = fix g 来达到这个效果。比如，我们能定义阶乘函数为：

```
g = λfct. λn. if realeq n c0 then c1 else (times n (fct (prd n)));
factorial = fix g;
```

图 5.2 说明在求值时项 factorial c_3 将发生什么。使这个演算起作用的关键事实是fct n $\longrightarrow^*$ g fct n。即 fct 是一种"自重复运算符"，当它运用到一个参数时，将它自己和 n 作为参数提供给 g。在 g 的第一个参数出现在 g 的体中时，我们将得到 fct 的另一个拷贝，对于这个拷贝，当运用到一个参数时，又将它自己和这个参数提供给 g，如此下去。每次我们用 fct 做一个递归调用，将展开 g 体的更多拷贝，并且提供给它 fct 的一个新拷贝，而这个新拷贝再次准备展开。

```
     factorial c3
=    fix g c3
→    h h c3
     其中 h = λx. g (λy. x x y)
→    g fct c3
     其中 fct = λy. h h y
→    (λn. if realeq n c0
             then c1
             else times n (fct (prd n)))
           c3
→    if realeq c3 c0
       then c1
       else times c3 (fct (prd c3))
→*   times c3 (fct (prd c3))
→*   times c3 (fct c2')
     其中 c2' 行为等价于 c2
→*   times c3 (g fct c2')
→*   times c3 (times c2' (g fct c1'))
     其中 c1' 行为等价于 c1
     (重复计算 g fct c2')
→*   times c3 (times c2' (times c1' (g fct c0')))
     其中 c0' 行为等价于 c0
     (类似)
→*   times c3 (times c2' (times c1' (if realeq c0' c0 then c1
                                      else ...)))
→*   times c3 (times c2' (times c1' c1))
→*   c6'
       其中 c6' 行为等价于 c6
```

图 5.2 factorial c_3 的求值过程

5.2.9　练习[★]：为什么在 g 的定义中我们用一个原语 if，而不是 Church 布尔式上的 Church 布尔函数 test？证明如何用 test 而不是 if 定义函数 factorial。

5.2.10　练习[★★]：定义一个函数 churchnat，能将一个原始自然数转换到相应的 Church 数值。

5.2.11　练习[推荐，★★]：用 fix 和练习(5.2.8)中列表的形式写出一个函数，它对 Church 数值列表中数值求和。

表示

在讨论 lambda 演算的形式定义之前，我们应当考虑最后的一个问题是：Church 数值表示普通自然数究竟意味着什么？

为回答这个问题，首先需要回忆一下什么是普通自然数。有几种(等价的)方法定义自然数；我们选择的方法是(参见图 3.2)定义自然数：

- 一个常量 0
- 一个将自然数映射到布尔值的运算 iszero
- 两个运算，succ 和 pred，将自然数映射到自然数

算术运算的行为是由图 3.2 中的求值规则所定义的。这些规则告诉我们，比如，3 是 2 的后继，iszero 0 为真等。

自然数的 Church 编码将这些元素表示为一个 lambda 项(即一个函数)：

- 项 c_0 表示自然数 0。

  如上述所见，也存在自然数作为项的"非典型表示"。比如，$\lambda s.\lambda z.(\lambda x.x)z$ 的行为等价于 c_0，也表示 0。
- 项 scc 和 prd 表示算术运算 succ 和 pred，也就是说，如果 t 是自然数 n 的一个表示，则 scc t 求值为 n+1 的一个表示，并且 prd t 求值为 n-1 的一个表示(或者 0 的表示，如果 n 是 0)。
- 项 iszro 表示运算 iszero，就是说，如果 t 是 0 的一个表示，则 iszro t 求值结果为 true①，如果 t 表示不是 0 的自然数，则 iszro t 求值结果为 false。

总之，假定我们有一整个程序来做自然数的复杂计算并产生一个布尔结果。如果用表示自然数和运算的 lambda 项替换所有的自然数和运算，将得到相同的结果。这样，借助于它们对程序整个结果的影响，在真实的自然数和它们的 Church 数值表示之间没有很明显的差别。

5.3　形式性

接下来，将更详细地考虑 lambda 演算的语法和操作语义。我们需要的大部分结构类似于在第 3 章中所见的(为避免逐字重复这个结构，在这里只提及不含布尔值或自然数的纯 lambda 演算)。然而，用一个项代换一个变量的操作存在一些惊人的困难。

---

① 严格地讲，如我们所定义的那样，iszero t 求值为 true 作为另一个项的一个表示，但让我们省略这个差别以简化讨论。类似地，可以解释在什么意义下 Church 布尔式表示真实的布尔值。

语法

如在第 3 章中一样,定义项的抽象语法应当看做是一个归纳定义的抽象语法树集合的简写形式。

5.3.1 定义[项]:设$\mathcal{V}$是一个变量名的可数集合。项的集合是最小的集合$\mathcal{T}$,若满足:

1. 对每个 $x \in \mathcal{V}$, $x \in \mathcal{T}$。
2. 如果 $t_1 \in \mathcal{T}$ 并且 $x \in \mathcal{V}$,则 $\lambda x.t_1 \in \mathcal{T}$。
3. 如果 $t_1 \in \mathcal{T}$ 并且 $t_2 \in \mathcal{T}$,则 $t_1 t_2 \in \mathcal{T}$。

一个项 t 的长度的定义与定义(3.3.2)中算术表达式的长度的定义相同。更有趣的是,我们能给出一个简单归纳来定义一个 lambda 项中自由变量的集合。

5.3.2 定义:一个项 t 的自由变量集合,记为 $FV(t)$,定义如下:

$$
\begin{aligned}
FV(x) &= \{x\} \\
FV(\lambda x.t_1) &= FV(t_1) \setminus \{x\} \\
FV(t_1\ t_2) &= FV(t_1) \cup FV(t_2)
\end{aligned}
$$

5.3.3 练习[★★]:给出详细的证明:对每个项 t, $|FV(t)| \leq size(t)$。

代换

仔细想想会发现实际上代换的操作是非常棘手的。在本书中,我们将采用两个不同的定义,每个针对不同的目的。在本节中引入的第一个定义比较紧致和直观,适合讨论例子以及数学定义和证明。在第 6 章中引入的第二个定义,符号更多,依赖于项的 de Bruijn 表示,其中用数值下标来代替命名的变量,但这个定义更适合在以后章节中讨论具体的 ML 实现。

先提几个错误的代换想法,这对我们理解代换是有启发作用的。首先,给出最初级的递归定义(形式地,定义一个函数$[x \mapsto s]$,对它的参数 t 进行归纳):

$$
\begin{aligned}
[x \mapsto s]x &= s \\
[x \mapsto s]y &= y \qquad\qquad \text{if } x \neq y \\
[x \mapsto s](\lambda y.t_1) &= \lambda y.\ [x \mapsto s]t_1 \\
[x \mapsto s](t_1\ t_2) &= ([x \mapsto s]t_1)\ ([x \mapsto s]t_2)
\end{aligned}
$$

这个定义对大部分例子是正确的。比如,它给出:

$$[x \mapsto (\lambda z.\ z\ w)](\lambda y.x) = \lambda y.\lambda z.\ z\ w,$$

这与我们关于如何代换的想法是一致的。但如果不仔细选择囿变量名称的话,定义就不正确了。比如:

$$[x \mapsto y](\lambda x.x) = \lambda x.y$$

这与关于函数抽象的基本想法不符,这个想法是:囿变量的命名是无关紧要的,尤其当我们用 λx.x,λy.y 或 λfranz.franz 表示恒等函数时是一致的。如果它们在代换下的行为不同,那么它们将在归约下的行为也会不同,这将出现问题。

显然,在代换的初级定义中犯的第一个错误是我们没有区别一个变量 x 在一个项 t 中的自由出现(在代换时应该代换的变量)和囿(受界定的)出现(在代换时不应该代换的变量)。当

我们见到一个在 t 项里面抽象绑定的 x，代换操作就应该停止。这产生了下一个定义：

$$
\begin{array}{lll}
[x \mapsto s]x & = & s \\
[x \mapsto s]y & = & y \qquad\qquad \text{if } y \neq x \\
[x \mapsto s](\lambda y.t_1) & = & \begin{cases} \lambda y.\ t_1 & \text{if } y = x \\ \lambda y.\ [x \mapsto s]t_1 & \text{if } y \neq x \end{cases} \\
[x \mapsto s](t_1\ t_2) & = & ([x \mapsto s]t_1)\ ([x \mapsto s]t_2)
\end{array}
$$

这个定义比上一个定义好，但仍然不是正确的定义。比如，考虑在项 $\lambda z.x$ 中用项 z 代换变量 x 时，将有：

$$[x \mapsto z](\lambda z.x) = \lambda z.z$$

这次，我们犯了一个几乎相反的错误：将常量函数 $\lambda z.x$ 变成一个恒等函数。这种情况的出现是因为我们碰巧选择 z 作为常量函数中囿变量的名称，因此，错误出现了。

当简单地代换一个项 s 到一个项 t，在项 s 中自由变量变成受界定的现象称为变量捕捉。为了避免这种现象的出现，我们需要确保 t 的受界定变量的名称不同于 s 的自由变量的名称。正确处理这一点的代换操作称为避免捕捉代换（这几乎就是正确"代换"的意思）。在抽象情况的第二个子句中加上一个附加条件，就能得到所需要的结果：

$$
\begin{array}{lll}
[x \mapsto s]x & =' & s \\
[x \mapsto s]y & = & y \qquad\qquad \text{if } y \neq x \\
[x \mapsto s](\lambda y.t_1) & = & \begin{cases} \lambda y.\ t_1 & \text{if } y = x \\ \lambda y.\ [x \mapsto s]t_1 & \text{if } y \neq x \text{ and } y \notin FV(s) \end{cases} \\
[x \mapsto s](t_1\ t_2) & = & ([x \mapsto s]t_1)\ ([x \mapsto s]t_2)
\end{array}
$$

我们几乎有了代换的正确定义：代换的这个定义在它做任何事情都是对的。问题是最后的方法将代换变成一个部分操作。比如，新定义对 $[x \mapsto y\ z](\lambda y.\ x\ y)$ 不产生任何结果：被代换项的囿变量 y 不等于 x，但它自由出现在 (y z) 中，因而，这个定义中没有一条子句可以使用。

在类型系统和 lambda 演算的文献中，解决这个问题的共同办法是假定项"在囿变量的重新命名下"是相同的（Church 用术语 alpha 转化表示在一个项中协调地重新命名一个囿变量的操作。这个术语仍然使用——我们也能说项"在 alpha 转化下"是相同的）。

5.3.4　约定：只是囿变量名称不同的项在所有上下文中可交替使用。

实际上这意味着任何 λ 囿变量的名称在方便时可以改变成另一个名称（在 λ 的体中统一改变）。比如，如果我们要计算 $[x \mapsto y\ z](\lambda y.\ x\ y)$，首先重写 $(\lambda y.\ x\ y)$，比如说，$(\lambda w.\ x\ w)$。然后计算 $[x \mapsto y\ z](\lambda w.\ x\ w)$，得出 $(\lambda w.\ y\ z\ w)$。

这个约定使得代换操作"尽可能好"，因为只要当我们发现要运用它到对它来说是无定义的参数时，如有必要可以重新命名参数，使得附加条件满足。的确，采用这个约定之后，可以更加简洁地形式化代换的定义。抽象的第一个子句可以省略，因为我们总是假定（如有必要，重新命名）囿变量 y 是不同于 x 和 s 的自由变量。于是得到下面的定义形式。

5.3.5　定义[代换]：

$$
\begin{array}{lll}
[x \mapsto s]x & = & s \\
[x \mapsto s]y & = & y \qquad\qquad \text{if } y \neq x \\
[x \mapsto s](\lambda y.t_1) & = & \lambda y.\ [x \mapsto s]t_1 \qquad \text{if } y \neq x \text{ and } y \notin FV(s) \\
[x \mapsto s](t_1\ t_2) & = & ([x \mapsto s]t_1)\ ([x \mapsto s]t_2)
\end{array}
$$

操作语义

图 5.3 总结了 lambda 项的操作语义。这里值的集合比在算术表达式中的情况更加有趣。因为(按值调用)求值到达一个 lambda 时停止,所以得出的值可以是任意 lambda 项。

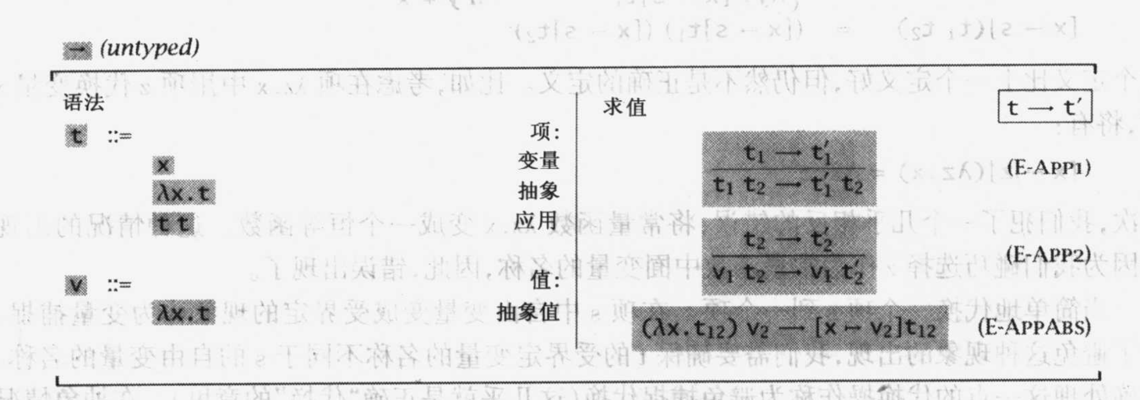


图 5.3　无类型 lambda 演算(λ)

求值关系出现在图的右边一列中。如同在算术表达式的求值一样,这里也有两种规则:计算规则 E-AppAbs 和同一规则 E-App1,E-App2。

注意在这些规则中元变量的选择如何帮助控制求值的次序。因为 v_2 可取任何值,规则 E-AppAbs 的左端可匹配任何右端是一个值的应用。类似地,规则 E-App1 可运用于任何左端不是一个值的应用,因为 t_1 可以匹配任何项,而前提要求 t_1 能做一步求值。另一方面,E-App2 不会激活,除非左端是一个能囿界于值元变量 v 的值。总之,这些规则完全确定了一个应用 $t_1 t_2$ 的求值次序:我们首先用 E-App1 归约 t_1 到一个值,然后用 E-App2 归约 t_2 到一个值,最后用E-Appabs 执行应用本身。

5.3.6　练习[★★]:采用这些规则来描述求值的其他 3 类策略:完全 beta 归约、规范序和懒惰求值。

注意,在纯 lambda 演算中,lambda 抽象是惟一可能的值,因此,如果我们到达这样一个状态:E-App1 成功地将 t_1 归约到一个值,则这个值必定是一个 lambda 抽象。如果加入其他构造函数,如原始布尔式到语言中,这个结论将不会成立,因为这些构造引入的值不是抽象形式。

5.3.7　练习[★★ ↛]:练习(3.5.16)中给出了布尔式和算术表达式的操作语义的另一种表示形式,其中定义受阻项使求值结果为特殊常量 wrong。将这个语义扩展到 **λNB**。

5.3.8　练习[★★]:练习(4.2.2)中引入了算术表达式求值的“大步”形式,其中基本求值关系是“项 t 求值为一个最后结果 v,说明如何在大步形式中形式化 lambda 项的求值规则。

5.4　注释

无类型 lambda 演算是 Church 和他的同事在 20 世纪 20 年代至 20 世纪 30 年代提出的(Church,1941)。无类型 lambda 演算的标准读物是 Barendregt(1984);Hindley 和 Seldin(1986)虽

然并不全面，但容易理解。在“Handbook of Theoretical Computer Science”中 Barendregt(1990)的文章是一个简单的综述。lambda演算方面的资料也能在许多函数式程序语言（如 Abelson 和 Sussman，1985；Friedman，Wand 和 Haynes，2001；Peyton Jones 和 Lester，1992）和程序语言语义（如 Schmidt，1986；Gunter，1992；Winskel，1993；Mitchell，1996）的教材中找到。Böhm 和 Berarducci (1985)介绍了一个编码一类数据结构为 lambda 项的系统方法。

尽管如此，Curry 否认是他提出的 currying 想法。这个想法通常应归为 Schönfinkel(1924)，但还有许多 19 世纪的数学家，如 Frege 和 Cantor 等。

除了作为一个逻辑来使用之外，这个系统的确可以有其他应用。

——Alonzo Church，1932

第 6 章　项的无名称表示[①]

在上一章中，我们讨论了“在囿变量重新命名下”的项，引入了一个一般性的约定：囿变量可以在任何时候重新命名以保证代换正确或因为某种理由一个新的名称更适合而需要这样做。实际上，我们想怎样拼写一个囿变量的名称都可以。这个约定在讨论基本概念和表达证明时是可以的，但在构建一个实现时需要对每个项选择单一表示；特别是必须决定变量的出现将如何表示。这里有几种方法：

1. 如目前为止所做的一样，我们能以符号形式表示变量，但不用考虑囿变量的隐式重新命名约定，而采用一个操作，为避免捕捉，在必要时显式地用“新”的名称替代囿变量。
2. 我们能用符号表示变量，但必须引入一个一般性的条件：所有囿变量的名称必须互相不同，还要不同于我们可能用到的任何自由变量。这个约定（有时称为 Barendregt 约定）更加严格，因为它不允许在任意时刻“随意”重新命名。然而，它在代换（或 beta 归约）下不稳定：因为代换会复制被代换的项，所以容易构造一个例子，它的代换结果是一个项，且项中某个 λ 抽象有相同的囿变量名称。这蕴涵着每个带有代换的求值步骤后面会有一个重新命名的步骤来保存不变式。
3. 设计某种不要求重新命名的变量和项的“典型”表示方式。
4. 通过引入一种机制，如显式代换（Abadi，Cardelli，Curien 和 Lévy，1991a）来避免一次代换。
5. 通过基于组合式的语言，如组合逻辑（Curry 和 Feys，1985；Barendregt，1984）（一个基于组合式而不是基于过程抽象的 lambda 演算）或 Backus 的函数式语言 FP（1978），来避免变量。

每个方法都有它的支持者，如何选择它们是值得推敲的问题（在严格的编译器实现中，还要考虑性能问题，但我们这里不关心这一点）。我们选择第 3 个，因为在我们讨论后面更复杂的实现时，这种方法更容易扩展。主要因为当它实现错误时，会出现灾难性的失败而不是难以察觉的失败，这样错误更容易被发现并及时改正。相比之下，命名变量实现中出现的错误可以在它们引入之后的几个月或几年中也不被发现。我们的形式化方法采用著名的 Nicolas de Bruijn（1972）技术。

6.1　项和上下文

de Bruijn 的想法是可以更直接地表示项（如果有任何缺点的话那就是可读性差）将变量出现直接指向它们的绑定器，而不是按名称引用它们。这能用自然数替代有名变量来实现，其中数 k 表示“第 k 个封闭 λ 所囿界的变量”。比如，普通项 λx. x 对应于无名称项 λ.0，而 λx. λy.

---

① 本章中讨论的系统是纯无类型 lambda 演算，λ（参见图 5.3）。相关的 OCaml 实现是 fulluntyped。

x(yx)对应于 λ.λ.1(0 1)。无名称项有时也称为 de Bruijn 项，并且项中的数值变量称为 de Bruijn 索引[①]。编译者可以用术语"静态距离"表示相同概念。

6.1.1　练习[★]：对每个下面的组合式：

```
c0 = λs. λz. z;
c2 = λs. λz. s (s z);
plus = λm. λn. λs. λz. m s (n z s);
fix = λf. (λx. f (λy. (x x) y)) (λx. f (λy. (x x) y));
foo = (λx. (λx. x)) (λx. x);
```

写出相应的无名称项。

形式地，我们定义无名称项的语法几乎和普通项的语法定义(5.3.1)一样。惟一的区别是我们需要记住：每个项可能含有的自由变量数目，以区别没有自由变量项的集合(称为 0 项)和至多含有一个自由变量的项的集合(1 项)。

6.1.2　定义[项]：设$\mathcal{T}$是最小的集簇$\{\mathcal{T}_0, \mathcal{T}_1, \mathcal{T}_2, \cdots\}$，使得：

1. 当 $0 \leq k < n$，$k \in \mathcal{T}_n$。
2. 如果 $t_1 \in \mathcal{T}_n$ 并且 $n > 0$ 则 $\lambda.t_1 \in \mathcal{T}_{n-1}$。
3. 如果 $t_1 \in \mathcal{T}_n$ 并且 $t_2 \in \mathcal{T}_n$ 则 $(t_1\, t_2) \in \mathcal{T}_n$。

注意这是一个标准的归纳定义，只是我们定义的是自然数索引的集簇，而不是单个集合。每个$\mathcal{T}_n$ 的元素称为 n 项。

$\mathcal{T}_n$ 的元素是至多含有 n 个自由变量的项，标记为 $0, \cdots, n-1$：一个给定$\mathcal{T}_n$ 的元素不一定有这么多自由变量，或根本不含有自由变量。比如，当 t 封闭时，对每个 n，t 都可以是$\mathcal{T}_n$ 的元素。

注意每个(封闭的)普通项只有一个 de Bruijn 表示，两个普通项在囿变量的取模重新命名下是等价的，当且仅当它们有相同的 de Bruijn 表示。

为了处理含自由变量的项，需要定义一个命名上下文的概念。比如，假定我们要表示 λx. y x 为一个无名称项。我们知道对 x 做什么，但不知道 y 的绑定器，因此不清楚这个绑定器可能在"多远的"地方，并且不知道指派给它什么数。一个解决方法是一次对所有的自由变量指派一个 de Bruijn 索引(称为一个命名上下文)，并且在需要选择自由变量的数时，保持一致地使用这个指派。比如，假定我们选择在下面的命名上下文中工作：

$$
\begin{aligned}
\Gamma \quad = \quad & x \mapsto 4 \\
& y \mapsto 3 \\
& z \mapsto 2 \\
& a \mapsto 1 \\
& b \mapsto 0
\end{aligned}
$$

则将 x(y z)表示为 4(3 2)，而将 λw.y w 表示为 λ.4 0 且 λw.λa.x 为 λ.λ.6。

因为变量出现在 Γ 中的次序决定它们的数值索引，我们能紧致地将它表示成一个序列。

6.1.3　定义：假定 x_0 到 x_n 是$\mathcal{V}$的变量名称。命名上下文 $\Gamma = x_n, x_{n-1}, \cdots, x_1, x_0$ 指派每个

① 注意读法：与"de Bruijn"第 2 个发音相近的英语单词是"brown"，而不是"broyn"。

x_i,一个 de Bruijn 索引为 i。注意序列最右边的变量的索引为 0;这与在转换一个命名项到无名称形式时,我们从右到左计数 λ 界定者的方法是一致的。用 $dom(\Gamma)$表示在 Γ 中提到的变量名称的集合$\{x_n, \cdots, x_0\}$。

6.1.4 练习[★★★ ↛]:用定义(3.2.3)的形式给出 n 项集合的构造,并且证明[如同在命题(3.2.6)中那样]它与上面的构造等价。

6.1.5 练习[推荐,★★★]:

1. 定义一个函数 $removenames_\Gamma(t)$,它取一个命名上下文 Γ 和一个普通项 t(使得 $FV(t) \subseteq dom(\Gamma)$),产生相应的无名称项。
2. 定义一个函数 $restorenames_\Gamma(t)$,使得对一个无名称项 t 和一个命名上下文 Γ,产生一个普通项[为此,应"标示"在 t 中被抽象所囿界的变量的名称。可以假设在 Γ 中的名称是两两不同的,并且变量名称的集合$\mathcal{V}$是有序的,这样就可以说"选择没有在 $dom(\Gamma)$ 中出现的第一个变量名称"]。

这对函数应该具有下列性质:对任何无名称的项 t:

$removenames_\Gamma(restorenames_\Gamma(t)) = t$

并且类似地,在囿界变量的重新命名下,对任何普通项 t:

$restorenames_\Gamma(removenames_\Gamma(t)) = t$;

严格地讲,称"某个 $t \in \mathcal{T}$"是没有意义的,我们总需要说明 t 可以有多少个自由变量。实际上,通常在头脑中有某个固定的命名上下文 Γ;我们还会滥用记号,用 $t \in \mathcal{T}$ 表示$t \in \mathcal{T}_n$,其中 n 是 Γ 的长度。

6.2 移位和代换

接下来是对无名称项定义一个代换操作($[k \mapsto s]t$)。为此,我们需要一个辅助操作,称为"移位",它将一个项中的自由变量的索引重新编号。

当代换在一个 λ 抽象下进行,比如在$[1 \mapsto s](\lambda.2)$(即$[x \mapsto \lambda s](\lambda y.x)$,假定 1 是 x 在外层上下文中的索引),代换发生时所在的上下文变成比原先多一个变量;我们需要增加在 s 中自由变量的索引使得它们在新的上下文中仍然引用相同的名称。但需要非常小心:我们不能移位 s 中的每个变量索引一个单位,因为这也将 s 中的囿变量移位。比如,如果$s = 2(\lambda.0)$(即 $s = z(\lambda w.w)$,假定 2 是 z 在外层上下文中的索引),我们需要移位这个 2 而不是 0。下面的移位函数采用一个"截"参数 c 来控制哪个变量应该移位。从 0 开始(意味着所有的变量都要移位),移位函数在通过一个绑定器时每次增加 1。因此,当计算 $\uparrow^d_c(t)$,我们知道项 t 是来自于原始$\uparrow^d$中 c 个绑定器的内部。因此,t 中所有标识符 k,当 $k < c$ 时,k 是受原始参数囿界的,不应该移位。而当 $k \geq c$ 时,k 是自由的,可以移位。

6.2.1 定义[移位]:一个项 t 在截 c 上的 d 步移位,记为 $\uparrow^d_c(t)$,定义如下:

$$
\begin{aligned}
\uparrow_c^d(\mathtt{k}) &= \begin{cases}\mathtt{k} & \text{if } k < c\\ \mathtt{k}+d & \text{if } k \geq c\end{cases}\\
\uparrow_c^d(\lambda.\mathtt{t}_1) &= \lambda.\ \uparrow_{c+1}^d(\mathtt{t}_1)\\
\uparrow_c^d(\mathtt{t}_1\ \mathtt{t}_2) &= \uparrow_c^d(\mathtt{t}_1)\ \uparrow_c^d(\mathtt{t}_2)
\end{aligned}
$$

我们用 $\uparrow^d(\mathrm{t})$表示 $\uparrow_0^d(\mathrm{t})$。

6.2.2　练习[★]:

1. $\uparrow^2(\lambda.\lambda.1(0\ 2))$等于什么?
2. $\uparrow^2(\lambda.0\ 1(\lambda.0\ 1\ 2))$等于什么?

6.2.3　练习[★★ ↛]:证明如果 t 是一个 n 项,则 $\uparrow_c^d(\mathrm{t})$是一个$(n+d)$项。

现在我们准备定义代换算子[j ↦ s]t。当用代换时,通常对上下文中最后的变量(即 j = 0)的代换感兴趣,因为这是我们为了定义 beta 归约运算所需要的。然而,为了在一个正好是一个 λ 抽象的项中代换变量 0,需要能在它的体内代换索引为 1 的变量数。这样,代换的定义肯定对任意变量也成立。

6.2.4　定义[代换]:用一个项 s 在 t 项中对变量数 j 的代换,记为[j ↦ s]t,定义如下:

$$
\begin{aligned}
[\mathtt{j}\mapsto\mathtt{s}]\mathtt{k} &= \begin{cases}\mathtt{s} & \text{如果 } \mathtt{k}=\mathtt{j}\\ \mathtt{k} & \text{其他}\end{cases}\\
[\mathtt{j}\mapsto\mathtt{s}](\lambda.\mathtt{t}_1) &= \lambda.\ [\mathtt{j}+1\mapsto\uparrow^1(\mathtt{s})]\mathtt{t}_1\\
[\mathtt{j}\mapsto\mathtt{s}](\mathtt{t}_1\ \mathtt{t}_2) &= ([\mathtt{j}\mapsto\mathtt{s}]\mathtt{t}_1\ [\mathtt{j}\mapsto\mathtt{s}]\mathtt{t}_2)
\end{aligned}
$$

6.2.5　练习[★]:将下面的代换转换为无名称形式,假定全局上下文是 Γ = a,b,并且用上面的定义计算它们的结果。答案是否对应于 5.3 节中普通项代换的定义?

1. [b ↦ a] (b (λx.λy.b))
2. [b ↦ a (λz.a)] (b (λx.b))
3. [b ↦ a] (λb. b a)
4. [b ↦ a] (λa. b a)

6.2.6　练习[★★ ↛]:证明如果 s 和 t 是 n 项并且 j≤n,则[j ↦ s]t 是一个 n 项。

6.2.7　练习[★ ↛]:拿出一张纸,不看代换和移位的定义,默写这两个定义。

6.2.8　练习[推荐,★★★]:无名称项的代换定义应该与普通项代换的非形式定义一致。(1)需要什么定理来严格地验证这个对应?(2)证明它。

6.3　求值

为了定义无名称项上的求值关系,我们需要改变的惟一地方(因为这是惟一提起变量名称的地方)是 beta 归约规则,它必须使用新的无名称代换操作。

惟一有点微妙之处是,一个约式的归约"用尽了"囿变量:当我们将$((\lambda x.\,t_{12})\,v_2)$归约为$[x\mapsto v_2]t_{12}$时,囿变量 x 在这个过程中消失了。这样,考虑到 x 不再是上下文的一部分,我们

需要重新标记代换后的变量。比如：

$$(\lambda.1\ 0\ 2)\ (\lambda.0) \longrightarrow 0\ (\lambda.0)\ 1 \qquad (\text{not } 1\ (\lambda.0)\ 2)$$

类似地，我们在代换到 t_{12}之前需要移位 v_2 中的变量一个单位，因为 t_{12}是定义在一个比 v_2 更大的上下文中的。考虑到这些 beta 归约规则是下列形式：

$$(\lambda.t_{12})\ v_2 \longrightarrow \uparrow^{-1}([0 \mapsto \uparrow^{1}(v_2)]t_{12}) \qquad \text{(E-AppAbs)}$$

其他规则如同以前(参见图 5.3)。

6.3.1 练习[★]：我们是否需要注意在这个规则中的负移位可能产生包含负索引的不良形式的项？

6.3.2 练习[★★★]：de Bruijn 原文献中实际上提出了两个不同的项无名称表示：这里的 de Bruijn索引，它"从里到外"计数 lambda 绑定器，以及 de Bruijn 级，它"从外到里"计数 lambda 绑定器。比如，项 $\lambda x.(\lambda y.xy)x$ 用 de Bruijn 索引表示为 $\lambda.(\lambda.1\ 0)0$ ，用 de Bruijn 级表示为 $\lambda.(\lambda.0\ 1)0$。精确地定义该变式，并且证明一个项用索引和级的表示是同构的(即一个能从另一个惟一地复原)。

第 7 章　lambda 演算的一个 ML 实现[①]

在这一章中，我们基于第 4 章中算术表达式的解释器和第 6 章中变量绑定和代换的处理，构造无类型 lambda 演算的一个解释器。

通过直接将前面的定义翻译到 OCaml，就可以得到无类型 lambda 演算的一个可执行的求值器。如同在第 4 章一样，我们将只说明核心算法，忽略词法分析、语法分析和打印等问题。

7.1　项和上下文

我们能直接转换定义(6.1.2)得到一个数据类型来表示项的抽象语法树：

```
type term =
    TmVar of int
  | TmAbs of term
  | TmApp of term * term
```

一个变量的表示是一个数——它的 de Bruijn 索引。一个抽象的表示只带有一个表示抽象体的子项。一个应用携带有被应用的两个子项。

实际用于实现中的定义将带有更多的信息。首先，如同以前，用记录了项原来位置的类型为 info 的元素来注释每个项是很有用的，这样错误打印程序可以引导用户(或自动产生用户文档编辑器)来精确定位错误出现的地方。

```
type term =
    TmVar of info * int
  | TmAbs of info * term
  | TmApp of info * term * term
```

其次，为了调试，在每个变量结点上加上一个附加的数作为一致检查是很有帮助的。约定第二个数将总是含有变量出现的上下文的总长度。

```
type term =
    TmVar of info * int * int
  | TmAbs of info * term
  | TmApp of info * term * term
```

当一个变量被打印，我们将验证这个数是否对应于当前上下文的实际长度；如果不是，则在某处丢掉了一个移位操作。

最后改进的工作也是关于打印的。尽管内部用 de Bruijn 索引表示项，但明显这不是呈现给用户的：我们应该在语法分析时将普通的表示转换为无名称项，并且在打印时转换回普通的表示。这样做没有什么难度，但不应该完全不加思索地进行(比如，对变量的名称产生完全新鲜的符号)，因为打印项中的囿变量的名称与原来程序中的名称将会没有任何关系。为此，可以通过用一个字符串暗示囿变量的名称来注释每一个抽象。

① 本章主要讨论的系统是纯无类型的 lambda 演算(参见图 5.3)。相关的实现是 untyped。fulluntyped 的实现包含了数和布尔值等扩展。

```
type term =
    TmVar of info * int * int
  | TmAbs of info * string * term
  | TmApp of info * term * term
```

项上的基本操作(特别是代换)与这些字符串没有任何关系:它们只是出现在原来的形式中,不需要进行名称冲突、捕捉等检查。当打印程序需要生成一个囿变量的一个新名称时,它首先企图用所提供的暗示;如果这与在当前上下文中已经使用的名称冲突,它试着用类似的名称,增加质数直到找到当前没有被使用的名称为止。这确保了打印的项,取几个质数的模后仍能接近用户所期望的。

打印程序本身看起来像这样:

```
let rec printtm ctx t = match t with
    TmAbs(fi,x,t1) →
      let (ctx',x') = pickfreshname ctx x in
      pr "(lambda "; pr x'; pr ". "; printtm ctx' t1; pr ")"
  | TmApp(fi, t1, t2) →
      pr "("; printtm ctx t1; pr " "; printtm ctx t2; pr ")"
  | TmVar(fi,x,n) →
      if ctxlength ctx = n then
        pr (index2name fi ctx x)
      else
        pr "[bad index]"
```

采用数据类型 context:

```
type context = (string * binding) list
```

是一个字符串和相关的 binding 列。目前,绑定本身完全是平凡的:

```
type binding = NameBind
```

不带有任何有意义的信息。以后(在第 10 章中),我们将引入绑定类型的其他子句来找到与变量相关的类型假设和其他类似信息。

打印函数也依赖于几个低级函数:pr 将一个串发送给标准的输出流;ctxlength 返回一个上下文的长度;index2name 从变量的索引中查找变量的串名称。最有趣的函数是 pickfreshname,它取一个上下文 ctx 和一个串暗示 x,找类似于 x 的名称 x',使得 x'没有列在 ctx 中,将 x'加入到 ctx 形成一个新的上下文 ctx',并且作为一个序对返回 ctx'和 x'。

在本书网站提供的 untyped 实现中找到的打印函数比起这个更加复杂,主要考虑到两个问题:首先,它尽可能地不注意括号,根据约定应用为左结合并且抽象体尽可能地扩展到右边。其次,它产生低层美化打印模块的格式指令(OCaml Format 库)来决定断行和缩进。

7.2 移位和代换

移位的定义(6.2.1)几乎可以逐个符号地翻译为 OCaml:

```
let termShift d t =
  let rec walk c t = match t with
    TmVar(fi,x,n) → if x>=c then TmVar(fi,x+d,n+d)
                    else TmVar(fi,x,n+d)
  | TmAbs(fi,x,t1) → TmAbs(fi, x, walk (c+1) t1)
  | TmApp(fi,t1,t2) → TmApp(fi, walk c t1, walk c t2)
  in walk 0 t
```

这里,用内层函数 walk c t 来表示内部移位 $\uparrow_c^d(t)$。因为 d 不会改变,不需要每次调用 walk 时单独传递:在 walk 的变量情况中,当需要它时才用到 d 的外层绑定。用 termShift d t 表示最高层移位 $\uparrow^d(t)$(注意 termShift 本身不标记为 recursive,因为它仅调用 walk 一次)。

类似地,代换函数几乎直接由定义(6.2.4)转化而来:

```
let termSubst j s t =
  let rec walk c t = match t with
    TmVar(fi,x,n) → if x=j+c then termShift c s else TmVar(fi,x,n)
  | TmAbs(fi,x,t1) → TmAbs(fi, x, walk (c+1) t1)
  | TmApp(fi,t1,t2) → TmApp(fi, walk c t1, walk c t2)
  in walk 0 t
```

这里,在项 t 中用项 s 对标号为 j 的变量代换 $[j \mapsto s]t$,记为 termSubst j s t。与原来代换定义的惟一差别是,这里在 TmVar 情况下一次做 s 的所有移位,而不是每次遇到一个绑定器时移位 s 一个单位。这意味着参数 j 在每次对 walk 的调用中都是相同的,并且我们能在内层的定义中忽略它。

读者可能注意到了 termShift 和 termSubst 的定义是非常类似的,只是在到达一个变量时的动作不同。在本书网站上的 untyped 实现探讨了这个现象,将移位和代换操作表示为一个称为 tmmap 的更一般函数的特殊情况。给定一个项 t 和一个函数 onvar,tmmap onvar t 的结果是一个形式与 t 相同的项,其中每个变量用调用 onvar 在变量上的结果替代。这个符号表示技巧在更复杂的演算中可以省去许多重复;25.2 节将更详细地讨论这一点。

在 lambda 演算的操作语义中,用到代换的惟一地方是在 beta 归约规则中。如我们以前注意的那样,这个规则实际上执行几个操作:对囿变量所代换的项首先被上移位一个单位,然后做代换,再将整个结果下移位一个单位来说明囿变量已经用完。下面的定义将这一系列的步骤封装为:

```
let termSubstTop s t =
  termShift (-1) (termSubst 0 (termShift 1 s) t)
```

7.3 求值

如在第 3 章中,求值函数依赖于一个辅助谓词 isval:

```
let rec isval ctx t = match t with
    TmAbs(_,_,_) → true
  | _ → false
```

单步求值函数是求值规则的一个直接副本,除了将上下文 ctx 与该项一起传递。这个参数在当前的 eval1 函数中没有用上,但在以后更复杂的求值器会用到它。

```
let rec eval1 ctx t = match t with
    TmApp(fi,TmAbs(_,x,t12),v2) when isval ctx v2 →
      termSubstTop v2 t12
  | TmApp(fi,v1,t2) when isval ctx v1 →
      let t2' = eval1 ctx t2 in
      TmApp(fi, v1, t2')
  | TmApp(fi,t1,t2) →
```

```
        let t1' = eval1 ctx t1 in
        TmApp(fi, t1', t2)
    | _ ->
        raise NoRuleApplies
```

多步求值函数与以前相同,除了 ctx 的参数:

```
let rec eval ctx t =
  try let t' = eval1 ctx t
      in eval ctx t'
  with NoRuleApplies → t
```

7.3.1 练习[推荐,★★★ ↛]:用在练习(5.3.8)中引入的大步形式的求值来改变这个实现。

7.4 注释

本章对代换的处理,尽管对本书来说足够了,但远不是最终的方案。特别地,在我们这个求值器中的 beta 归约规则"急切"用参数值代换函数体中的囿变量。为了速度而不是为了简单性的函数式语言的解释器(和编译器)采用不同的策略:不是实际执行代换,而只是在一个称为 environment 的辅助数据结构中记录囿变量名称和变量值之间的关联,这个环境与被求值的项一同存在。当到达一个变量时,在当前环境中寻找它的值。这个策略可以模型化为将环境看做是一种显式代换——即将代换的机制从元语言移到对象语言中,使它成为求值器所处理的项的语法的一部分,而不是项上的一个外部操作。Abadi, Cardelli, Curien 和 Lévy (1991a)首先研究显式代换,并从此成为一个活跃的研究领域。

只因为实现了某件事情并不意味着理解了这件事情。

——Brian Cantwell Smith

第二部分　简 单 类 型

第 8 章 类型算术表达式[①]

在第 3 章中,我们用含布尔值和算术表达式的一个简单语言介绍了精确描述语法和求值的基本工具。现在回到这个简单的语言,并加入静态类型。类型系统本身几乎是平凡的,但它提供了一个可以引入本书将重复提到的概念的环境。

8.1 类型

回忆一下算术表达式的语法:

```
t ::=                                   项:
      true                              常量真
      false                             常量假
      if t then t else t                条件式
      0                                 常量零
      succ t                            后继
      pred t                            前驱
      iszero t                          零测试
```

我们在第 3 章中看到求值一个项的结果,要么是一个值:

```
v ::=                                   值:
      true                              真值
      false                             假值
      nv                                数值

nv ::=                                  数值:
      0                                 零值
      succ nv                           后继值
```

要么在某步受阻,即到达一个像 pred false 这样的项,对这个项没有求值规则可使用。

受阻项对应着无意义或错误的程序。我们因此希望能够在没有实际求值一个项之前知道求值将不会受阻。为此,需要能够区别结果是一个数值的项(因为只有它们能作为 pred, succ, iszero 的参数)和结果是一个布尔值的项(只有它们能作为一个条件式的条件)。我们引入两个类型,Nat 和 Bool 来区分这些项。在本书中将用元变量 S,T,U 等来表示类型。

称"一个项 t 有类型 T"(或"t 属于 T",或"t 是 T 的一个元素")是指 t"明显"求值为适当形式的值——这里的"明显"是指我们不需要对 t 做求值就能静态地看到它的值类型。比如,项 if true then false else true 为类型 Bool ,而 pred(succ(pred(succ 0)))为类型 Nat。尽管如此,项的类型分析仍是保守的,因为只用到了静态信息。这就意味着我们不能下结论:有些项如 if(iszero 0) then 0 else false 或 if true then 0 else false 总会有类型,尽管它们的求值不会受阻。

---

① 本章中讨论的系统是布尔值和数的类型演算(参见图 8.2)。相应的 OCaml 实现是 tyarith。

8.2 类型关系

算术表达式的类型关系,记为 t:T[①],定义为一个指派类型到项的推导规则的集合,图 8.1 和图 8.2 是有关于规则的介绍。如同在第 3 章一样,我们在两个不同的图中给出布尔值的规则和数的规则,因为以后将要分别引用它们。

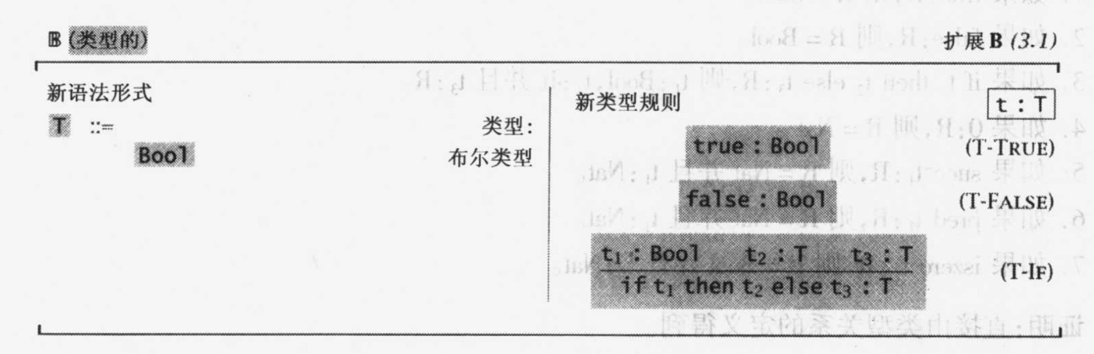


图 8.1　布尔(B)类型规则

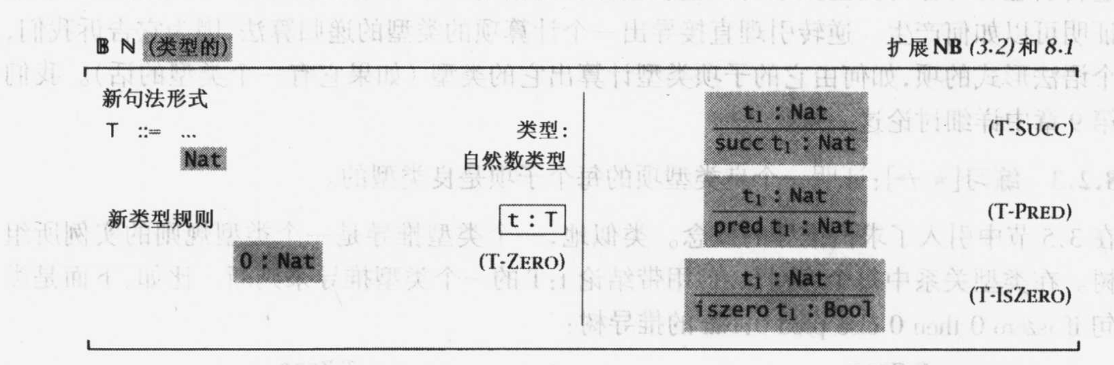


图 8.2　数(NB)的类型规则

图 8.1 中的规则 T-True 和 T-False 指派类型 Bool 给布尔常量 true 和 false。规则 T-If 基于它的子表达式的类型指派一个类型到一个条件表达式:条件 t_1 必须求值为一个布尔值,而 t_2 和 t_3 必须求值为相同类型的值。两次使用单个元变量 T 表示:if 的结果是 then 和 else 分支的类型,且可以是任何类型[要么是 Nat,要么是 Bool,要么(当我们考虑其他更有意义的演算时)可以是其他类型]。

图 8.2 中数的规则有一个相似的形式。T-Zero 指派类型 Nat 到常量 0。只要 t_1 有类型 Nat,T-Succ 就给项 succ t_1 指派类型 Nat。同样,T-Pred 和 T-IsZero 说明 pred 的参数有类型 Nat 时就产生一个 Nat,并且 iszero 的参数有类型 Nat 时就产生一个 Bool。

8.2.1　定义:形式地,算术表达式的类型关系是项和类型之间的最小二元关系,它满足图 8.1 和图 8.2 中所有规则的实例。则称,如果存在某个 T 使得 t:T 一个项 t 是可类型化的(或良类型的)。

① 常常采用符号∈,而不是":"。

当对类型关系进行推理时,我们将常常这样认为:"如果一个形为 succ t_1 的项是有类型的,则它的类型为 Nat"。下面的引理给出这个说法的一个纲要,直接由相应类型规则的形式得到。

8.2.2 引理[类型关系的逆转引理]:

1. 如果 true:R,则 R = Bool。
2. 如果 false:R,则 R = Bool。
3. 如果 if t_1 then t_2 else t_3:R,则 t_1:Bool,t_2:R 并且 t_3:R。
4. 如果 0:R,则 R = Nat 。
5. 如果 succ t_1:R,则 R = Nat 并且 t_1:Nat。
6. 如果 pred t_1:R,则 R = Nat 并且 t_1:Nat。
7. 如果 iszero t_1:R,则 R = Bool 并且 t_1:Nat。

证明:直接由类型关系的定义得到。

逆转引理有时称为类型关系的产生引理,因为,给定一个有效的类型语句,它说明这个逆转的证明可以如何产生。逆转引理直接导出一个计算项的类型的递归算法,因为它告诉我们,对每个语法形式的项,如何由它的子项类型计算出它的类型(如果它有一个类型的话)。我们将在第 9 章中详细讨论这一点。

8.2.3 练习[★ ↛]:证明一个良类型项的每个子项是良类型的。

在 3.5 节中引入了求值推导的概念。类似地,一个类型推导是一个类型规则的实例所组成的树。在类型关系中每个序对(t,T)用带结论 t:T 的一个类型推导来判断。比如,下面是类型语句 if iszero 0 then 0 else pred 0:Nat 的推导树:

$$\dfrac{\dfrac{\dfrac{}{\texttt{0 : Nat}}\ \text{T-Zero}}{\texttt{iszero 0 : Bool}}\ \text{T-IsZero} \qquad \dfrac{}{\texttt{0 : Nat}}\ \text{T-Zero} \qquad \dfrac{\dfrac{}{\texttt{0 : Nat}}\ \text{T-Zero}}{\texttt{pred 0 : Nat}}\ \text{T-Pred}}{\texttt{if iszero 0 then 0 else pred 0 : Nat}}\ \text{T-If}$$

换言之,语句是关于程序类型的形式断言,类型规则是语句之间的蕴涵关系,推导是基于类型规则的演绎。

8.2.4 定理[类型的惟一性]:每个项 t 至多有一个类型。即如果 t 是可类型化的,则它的类型是惟一的。此外,根据图 8.1 和图 8.2 中的推导规则只存在一个此类型的推导方式。

证明:直接对 t 做结构归纳,对每个情况利用逆转引理中的适当子句(加上归纳假设)。

在这一章简单类型系统的讨论中,每个项有一单个类型(如果它有类型的话),并且只有一个推导树来验证这个事实。以后(比如,在第 15 章中讨论带子类型的类型系统时)这些性质将不再满足:单个项可以有多个类型,并且一般可以有多种方法可以推导出一个给定的项有一个给定的类型。

类型关系的性质常常用对推导树做归纳来证明,就像求值关系的性质常常用对求值推导做归纳来证明一样。从下一节开始,我们将看到许多对类型推导做归纳的例子。

8.3　安全性 = 进展 + 保持

一个类型系统的最基本性质的状态是安全性(也称为可靠性):良类型项不会"出错"。我们已经知道如何形式化一个项出错的状态:它意味着到达了一个"受阻状态"[定义(3.5.15)],并不意味着这时为一个最后值,而是求值规则不能告诉我们下一步做什么。我们所要知道的是良类型项不会受阻。用两个步骤来证明这一点,这两个步骤常常称为进展和保持定理[1]。

进展:一个良类型项不会受阻(要么它是一个值,要么它根据求值规则做下一步)。

保持:如果一个良类型项做一步求值,则所得到的项也是良类型的[2]。

这些性质告诉我们一个良类型的项在求值过程中不会到达一个受阻状态。

为了证明进展定理,先说明几个关于类型 Bool 和 Nat 可能的典型形式(即这些类型的良类型值)。

8.3.1　引理[典型形式]:

1. 如果 v 是类型为 Bool 的一个值,则 v 要么为 true 要么为 false。
2. 如果 v 是类型为 Nat 的一个值,则根据图 3.2 中的文法,v 是一个数值。

证明:对部分(1),根据图 3.1 和图 3.2 中的语法,这个语言中的值能有 4 种形式:true,false,0 和 succ nv,其中 nv 是一个数值。前两个情况直接得到所需要的结果。后两个情况不会出现,因为我们假设 v 有类型 Bool,并且逆转引理的情况 4 和情况 5 告诉我们 0 和 succ nv 能有惟一的类型 Nat,而不是 Bool。部分(2)也是类似的。

8.3.2　定理[进展]:假定 t 是一个良类型的项(即对某个 T,有 t:T)。则要么 t 是一个值,要么存在某个 t′使得 t →t′。

证明:对 t:T 的一个推导做归纳证明。T-True,T-False 和 T-IsZero 情况下结论是直接得到的,因为在这些情况中 t 是一个值。对于其他情况,分别讨论如下:

情况 T-If:　$t = \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3$

$t_1 : \texttt{Bool}\quad t_2 : T\quad t_3 : T$

由归纳假设,要么 t_1 是一个值,要么存在某个 t_1' 使得 $t_1 \to t_1'$。如果 t_1 是一个值,则典型形式引理(8.3.1)确定它必然是 true 或是 false,在这种情况下,E-IfTrue 或 E-IfFalse 可应用于 t。另一方面,如果 $t_1 \to t_1'$,则由 E-If,$t \to \text{if } t_1' \text{ then } t_2 \text{ else } t_3$。

情况 T-Succ:　$t = \texttt{succ}\ t_1\qquad t_1 : \texttt{Nat}$

由归纳假设,要么 t_1 是一个值,要么存在某个 t_1' 使得 $t_1 \to t_1'$。如果 t_1 是一个值,则由典型形式引理,它必然是一个数值,因此 t 也是一个数值。另一方面,如果 $t_1 \to t_1'$ 则由E-Succ,$\text{succ } t_1 \to \text{succ } t_1'$。

---

① 标语"安全性是进展加保持"(利用一个典型形式引理)是 Harper 明确提出的;Wright 和 Felleisen(1994)提出一个类似的说法。

② 在我们讨论的大部分类型系统中,求值不仅保持良定性,同时也保持项的类型。而在某些系统中,类型在求值期间是可以改变的。比如,在带子类型的系统(参见第 15 章)中,类型在求值期间可以变得越来越小(具有更多信息)。

情况 T-PRED: $t = \texttt{pred}\ t_1 \quad t_1 : \texttt{Nat}$

由归纳假设,要么 t_1 是一个值,要么存在某个 t_1' 使得 $t_1 \rightarrow t_1'$。如果 t_1 是一个值,则由典型形式引理,它必然是一个数值,即要么是 0,要么对某个 nv_1 是 succ nv_1,并且规则 E-PredZero 或 E-PredSucc 可应用于 t。另一方面,如果 $t_1 \rightarrow t_1'$,则由 E-Pred,pred $t_1 \rightarrow$ pred t_1'。

情况 T-ISZERO: $t = \texttt{iszero}\ t_1 \quad t_1 : \texttt{Nat}$

与前类似。

类型在求值下保持的证明对这个系统也可直接得到。

8.3.3 定理[保持]: 如果 t:T 并且 $t \rightarrow t'$,则 t':T。

证明: 对 t:T 的一个推导做归纳证明。在归纳的每一步,我们假定所需要证明的性质对所有子推导成立(即,如果 s:S 并且 $s \rightarrow s'$,则只要由当前的一个子推导证明出 s:S,就有 s':S),接下来对推导中的最后规则做情况分析(我们只证明其中几个情况,其他是类似的)。

情况 T-TRUE: $t = \texttt{true} \quad T = \texttt{Bool}$

如果在推导中的最后规则是 T-True,则由这个规则的形式知道:t 必然是一个常量 true,并且 T 是 Bool。但 t 是一个值,所以不可能存在某个 t' 使得 $t \rightarrow t'$,并且如果是这种情况,定理就毫无意义了。

情况 T-IF: $t = \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \quad t_1 : \texttt{Bool} \quad t_2 : T \quad t_3 : T$

如果在推导中的最后规则是 T-If,则由这个规则的形式知道:必然对某些 t_1,t_2 和 t_3,t 必有形式 if t_1 then t_2 else t_3。也必然有含结论 t_1:Bool,t_2:T 和 t_3:T 的子推导。现在,看一下(参见图 3.1)左端带 if 的求值规则,发现有 3 个规则使得 $t \rightarrow t'$ 是可推导的:E-IfTrue,E-If-False 和E-If。我们分别考虑每种情况(省略情况 E-IfFalse,它与 E-IfTrue 类似)。

子情况 E-IFTRUE: $t_1 = \texttt{true} \quad t' = t_2$

如果 $t \rightarrow t'$ 用 E-IfTrue 可推导,则由这个规则的形式可以知道,必然 t_1 是 true,并且结果项 t' 是第二子表达式 t_2。这样就得证了,因为我们知道 t_2:T(由情况 T-If 的假设),这正是我们需要证明的。

子情况 E-IF: $t_1 \longrightarrow t_1' \quad t' = \texttt{if}\ t_1'\ \texttt{then}\ t_2\ \texttt{else}\ t_3$

由情况 T-If 的假设,有一个原来结论为 t_1:Bool 的类型推导的子推导。我们能运用归纳假设到这个子推导,得到 t_1':Bool。将这一点与(由情况 T-If 的假设)t_2:T 和 t_3:T 事实结合,我们能用规则 T-If 得出 if t_1' then t_2 else t_3:T,即 t':T。

情况 T-ZERO: $t = 0 \quad T = \texttt{Nat}$

不可能发生(由于与上述的 T-True 同样的理由)。

情况 T-SUCC: $t = \texttt{succ}\ t_1 \quad T = \texttt{Nat} \quad t_1 : \texttt{Nat}$

观察图 3.2 中的求值规则,我们知道只有一个规则——E-Succ,能用于推导 $t \rightarrow t'$。这个规则的形式告诉我们,$t_1 \rightarrow t_1'$。因为我们也知道 t_1:Nat,能运用归纳假设得到 t_1':Nat,由此可得到 succ t_1':Nat,即根据应用 T-Succ 规则,有 t':T。

8.3.4　练习[★★ ⇸]：重构这个证明使其对求值推导而不是对类型推导做归纳证明。

保持定理常常称为主题归约(或主题求值)，直观上理解为一个类型语句 t:T 可以看作为一个句子："t 有类型 T"。项 t 是这个句子的主题，且主题归约性质说明这个句子的真假值是在主题的归约下保持的。

不像类型的惟一性，在某些类型系统中成立，在某些系统中不成立，进展和保持将是我们考虑的所有类型系统的基本要求[①]。

8.3.5　练习[★]：求值规则 E-PredZero(参见图 3.2)有点违反直觉：我们可能觉得让零的前驱无定义更合理，而不是让零的前驱定义为零。能否在单步求值的定义中除去这个规则而做到这一点呢?

8.3.6　练习[推荐,★★]：有了主题归约性质之后，也会想知道相反的性质——主题扩张是否也成立。是否总存这种情况成立：如果 t →t′并且 t′:T 则 t:T? 如果成立，证明它；否则给出一个反例。

8.3.7　练习[推荐,★★]：假定我们的求值关系是在大步形式中定义，如练习(3.5.17)。类型安全性的直观性质如何能形式化?

8.3.8　练习[推荐,★★]：假定我们的求值关系增加一些规则来将无意义的项归约为一个明显的 wrong 状态，如在练习(3.5.16)中一样。现在如何形式化类型安全性?

在许多不同的领域从无类型到有类型的领域走过的道路多次重复，大部分又是为了相同的理由。

——Luca Cardelli 和 Peter Wegner(1985)

---

① 这些性质在有些语言中是不成立的。尽管如此，这些语言也认为是类型－安全的。比如，如果我们用小步形式来形式化 Java 的操作语义(Flatt, Krishnamurthi 和 Felleisen, 1998a; Igarachi, Pierce 和 Wadler, 1999)，这里给的形式的类型保持将不成立(详见第 19 章)。尽管如此，可以认为这是形式化的一个人为因素，而不是语言本身的缺点，因为它在语义的大步表示中类型保持根本不存在。

第 9 章　简单类型的 lambda 演算[1]

这一章将介绍在本书中讨论的类型语言系列中最基础的一个成员：Church(1940)和 Curry (1958) 的简单类型 lambda 演算。

9.1　函数类型

在第 8 章中，我们介绍了带两个类型算术表达式的一个简单静态类型系统：Bool(具有该类型的项求值为一个布尔值)和 Nat(具有该类型的项求值为一个数值)。不属于这两个类型的所谓"非良类型"项包括在求值时到达受阻状态的项(如 if 0 then 1 else 2)，以及实际上在求值时行为良好的项(如 if true then 0 else false)，这是因为我们的静态分类太保守了。

假设要构造一个包含布尔值(为简洁起见，将不考虑数值)和纯 lambda 演算原语的一个类似的类型系统。即要引入关于变量、抽象和应用的类型规则以保持类型安全[即满足类型保持和进展定理(8.3.2)和定理(8.3.3)]，且不是太保守，即它们可以给我们实际关心的大部分程序指派类型。

当然，由于纯 lambda 演算是 Turing 完备的，无法给出这些原语的精确类型分析。比如，无法在不实际运行复杂计算的情况下可靠地确定如下的程序：

```
if <复杂计算> then true else (λx.x)
```

是否产生一个布尔值或一个函数，及看到结果是 true 还是 false。通常，复杂计算可能发散，并且任何企图预测是否发散的类型检查器也将发散。

为扩展布尔值的类型系统使之包括函数，我们需要加入一个类型来分类这样的项，它的求值结果为一个函数。作为一个逼近，我们称这个类型为→。如果加入一个类型规则：

$$\lambda x.t : \to$$

使每个 λ 抽象类型，我们能分类像 λx.x 的简单项，也能分类像：if true then (λx.true) else (λx.λy.y)这样产生函数的复合项。

但这个粗略分析显然太保守：像 λx.true 和 λx.λy.y 这样的函数归为相同的类型→，而忽略了这样一个事实：应用第一个到 true 产生一个布尔值，而运用第二个到 true 产生另一个函数。一般地，为了给一个应用结果有用的类型，需要知道左端是一个函数外，还要知道：函数将返回什么类型。此外，为了确保当函数被调用时其行为良好，需要了解函数所期望的参数类型。为了得到这样的信息，我们用形为 $T_1 \to T_2$ 的类型的无限集合替换空类型→，其中函数类型 $T_1 \to T_2$ 说明每个分类函数希望获得类型为 T_1 的参数，并返回类型 T_2 的结果。

9.1.1　定义：类型 Bool 上的简单类型集合是由下列语法产生的：

---

[1] 本章中讨论的系统是带布尔值的简单类型 lambda 演算(参见图 9.1)。相应的 OCaml 实现是 fullsimple。

T ::=		类型：
	Bool	布尔类型
	T→T	函数类型

类型构造子→是右结合的——即表达式 $T_1 \to T_2 \to T_3$ 表示 $T_1 \to (T_2 \to T_3)$。

比如 Bool→Bool 是一个函数类型，将布尔型参数映射到一个布尔型结果。(Bool→Bool)→(Bool→Bool)或等价地，(Bool→Bool)→Bool→Bool 是一个函数类型，将布尔到布尔的函数作为参数，并且得出的结果也是布尔到布尔的函数。

9.2 类型关系

为了给像 λx.t 这样的抽象指派一个类型，当抽象应用于某个参数时需要计算会发生什么。下一个问题是：如何知道期望得出的参数是什么类型？存在两种可能：要么简单地用其参数需要的类型注释 λ 抽象，要么分析抽象体看看参数如何被使用，并由此演绎出它应有的类型。现在，选择第一种方案。记 $\lambda x{:}T_1.t_2$，而不是 λx.t，其中对囿变量的注释告诉我们假定参数类型是 T_1。

一般地，借助于项的类型注释来检查类型的语言称为显式类型化语言。我们要求类型检查器自己推导或重构这个信息的语言称为隐式类型化语言(在 λ 演算中，也用类型指派系统)。本书将大部分集中于显式类型化语言；隐式类型化语言将在第 22 章中讨论。

一旦我们知道抽象的参数类型，就知道了函数结果的类型将是体 t_2 的类型，其中 t_2 中的 x 假定表示类型为 T_1 的项。这可以由下列类型规则来表示：

$$\frac{x{:}T_1 \vdash t_2 : T_2}{\vdash \lambda x{:}T_1.t_2 : T_1 \to T_2} \qquad \text{(T-Abs)}$$

由于项可以包含嵌套 λ 抽象，我们一般将需要考虑几个这样的假设。这将改变类型关系，如从一个二元关系 t:T 到一个三元关系 $\Gamma \vdash t{:}T$，其中 Γ 是一个关于 t 中自由类型的假设集合。

形式地，一个类型化上下文(也称一个类型环境)Γ 是一个变量和它们类型的序列，并且逗号算子(comma)通过在右边加入一个新绑定来扩展 Γ。空上下文有时记为∅，但通常我们省略它，用$\vdash t{:}T$ 表示"封闭项 t 在空的假设集下有类型 T"。

为避免新绑定与出现在 Γ 中的任何绑定相混淆，要求选择的 x 名称不同于被 Γ 囿界的变量。由于我们的约定：由 λ 抽象囿界上的变量可以在方便时重新命名，这个条件在必要时通过重新命名囿变量总是可以满足的。Γ 可以看做是从变量到它们类型的一个有限函数。根据这一点，我们用 *dom*(Γ)表示由 Γ 界定的变量集合。

抽象的类型规则的一般形式为：

$$\frac{\Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.t_2 : T_1 \to T_2} \qquad \text{(T-Abs)}$$

其中前提比结论多一个假设。

变量的类型规则可以这样直接得到：一个变量可以有任何我们当前假定的类型：

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{(T-Var)}$$

前提 $x{:}T \in \Gamma$ 读做" 在 Γ 中假设 x 的类型为 T"。

最后，我们需要一个应用类型规则：

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \tag{T-App}$$

如果 t_1 求值为这样的一个函数，映射 T_{11} 中的参数到 T_{12} 中的结果（假设它的自由变量表示的值类型为 Γ 中假定的类型），并且如果 t_2 求值为 T_{11} 中的一个结果，则应用 t_1 到 t_2 的结果将是类型为 T_{12} 的一个值。

布尔常量和条件表达式的类型规则如前所述（参见图 8.1）。注意，尽管如此，在条件规则中的元变量 T：

$$\frac{\Gamma \vdash t_1 : \mathtt{Bool} \qquad \Gamma \vdash t_2 : T \qquad \Gamma \vdash t_3 : T}{\Gamma \vdash \mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3 : T} \tag{T-If}$$

能实例化为任何函数类型，允许分支是函数的类型条件式类型化[①]：

```
if true then (λx:Bool. x) else (λx:Bool. not x);
▸ (λx:Bool. x) : Bool → Bool
```

这些类型规则在图 9.1（为完整起见，还有语法和求值规则）中进行了总结。图中加有阴影的区域指明与无类型演算不同的新规则——新规则和加到老规则中的新部分。类似于处理布尔值和数值，我们将整个演算的定义分为两个部分：不带任何基类型的纯简单类型 lambda 演算（如图 9.1所示）和独立的布尔型规则集（参见图 8.1，我们必须将上下文加入到图中每个类型语句中）。

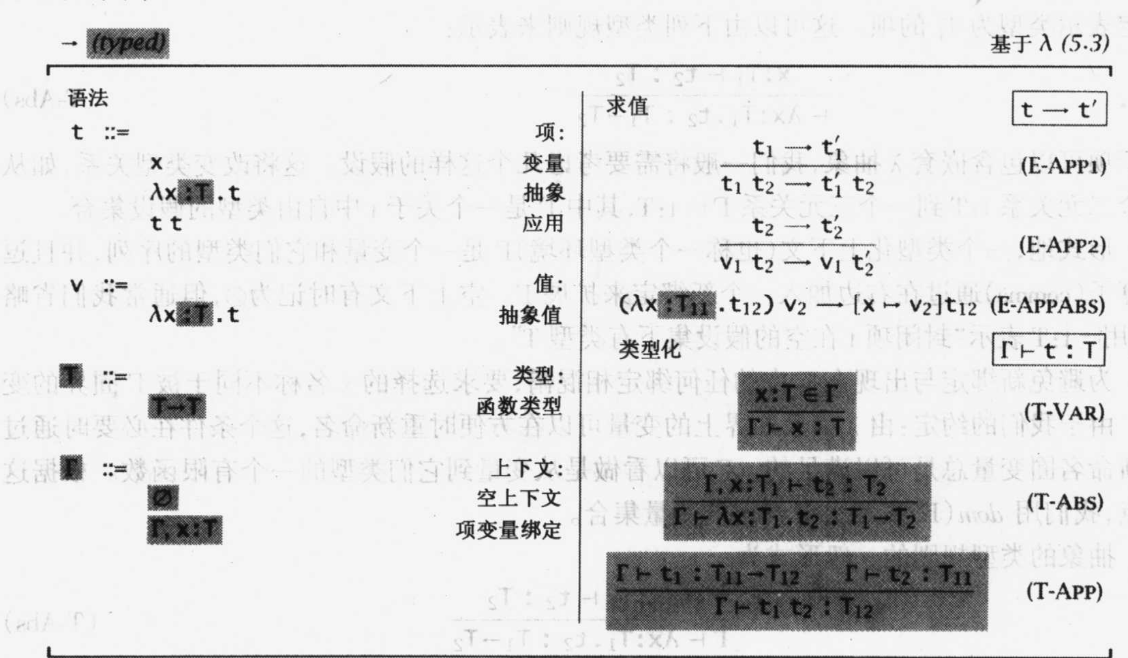


图 9.1　纯简单类型的 lambda 演算（$\lambda_{\to}$）

我们经常用符号 $\lambda_{\to}$ 表示简单类型的 lambda 演算（用相同的符号表示不同基类型的系统）。

---

① 从现在起，例子及其实现都将显示出执行结果及结果类型（当它们是显式时，有时会省略）。

9.2.1　练习[★]:不带基础类型的纯简单类型 lambda 演算实际是退化的,即它没有良定项。为什么?

如类型算术表达式所示,$\lambda_{\rightarrow}$的类型规则的实例可以组合成推导树。比如,下面的推导,说明项(λx:Bool.x)true 在空的上下文中有类型 Bool:

$$\dfrac{\dfrac{\dfrac{\texttt{x:Bool} \in \texttt{x:Bool}}{\texttt{x:Bool} \vdash \texttt{x : Bool}}\text{ T-VAR}}{\vdash \lambda\texttt{x:Bool.x : Bool}\to\texttt{Bool}}\text{ T-ABS} \qquad \dfrac{}{\vdash \texttt{true : Bool}}\text{ T-TRUE}}{\vdash (\lambda\texttt{x:Bool.x) true : Bool}}\text{ T-APP}$$

9.2.2　练习[★ ↛]:用画推导树的方法证明下列项有所指示的类型:

1. `f:Bool→Bool ⊢ f (if false then true else false) : Bool`

2. `f:Bool→Bool ⊢ λx:Bool. f (if x then false else x) : Bool→Bool`

9.2.3　练习[★]:找一个上下文 Γ 使得在此上下文中项 f x y 有类型 Bool 。你能否简单描述一下所有满足这个要求的上下文集合?

9.3　类型的性质

如在第 8 章中一样,在证明类型安全性之前需要几个基本引理。大部分引理类似于以前所见到的,我们只需要将上下文加入到类型关系中并将子句加入到每个 λ 抽象、应用和变量的证明中。惟一重要的新要求是类型关系的代换引理[参见引理(9.3.8)]。

首先,逆转引理说明了如何建立类型推导:每个语法形式的子句告诉我们"如果这个形式的项是良类型的,则它的子项必定有这些形式的类型"。

9.3.1　引理[类型关系的逆转]:

1. 如果 $\Gamma \vdash x:R$,则 $x:R \in \Gamma$。
2. 如果 $\Gamma \vdash \lambda x:T_1.t_2:R$,则对某个 R_2 使得 $\Gamma, x:T_1 \vdash t_2:R_2$,有 $R = T_1 \rightarrow R_2$。
3. 如果 $\Gamma \vdash t_1\,t_2:R$,则存在某个类型 T_{11} 使得 $\Gamma \vdash t_1:T_{11} \rightarrow R$ 并且 $\Gamma \vdash t_2:T_{11}$。
4. 如果 $\Gamma \vdash \text{true}:R$,则 $R = \text{Bool}$ 。
5. 如果 $\Gamma \vdash \text{false}:R$,则 $R = \text{Bool}$ 。
6. 如果 $\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3:R$,则 $\Gamma \vdash t_1:\text{Bool}$ 并且 $\Gamma \vdash t_2, t_3:R$。

证明:直接由类型关系的定义得出。

9.3.2　练习[推荐,★★★]:是否存在某上下文 Γ 和类型 T 使得 $\Gamma \vdash x\,x:T$? 如果存在,给出 Γ 和 T,并给出 $\Gamma \vdash x\,x:T$ 的类型推导;如果不存在,证明这一点。

在 9.2 节中,选择演算的显式类型化表示来简化类型检查的工作。这包括对在函数抽象中的囿变量加上类型注释,但其他地方不加。为什么这样就可以了呢? 一个答案是由"类型惟一性"定理给出的,这个定理说明良类型的项一一对应于它们的类型推导:类型推导能从项中惟一地恢复(反之也然)。事实上,这个对应十分直接,在某种意义上项与推导几乎没有区别。

9.3.3 定理[类型惟一性]:在一个给定的类型上下文 Γ 中,一个项 t(其所有自由变量在 Γ 的定义域中)至多有一个类型。即如果一个项是可类型化的,则它的类型是惟一的。此外,只有一个由产生类型关系的推导规则建立的类型化推导。

证明:留做练习。证明实际上可以直接写出,没有什么复杂性;但详细写出对"建立"类型关系的证明方面是一个好的实践。

对于书后面提到的许多类型系统,项与推导之间的这种简单对应将不再成立:单个项被指派几个类型,并且每个类型由几个类型推导判定。在这些系统中,这表明类型推导能有效地根据项来恢复是十分重要的。

下面的典型形引理告诉我们各种类型值的可能形式。

9.3.4 引理[典型形式]:

1. 如果 v 是一个类型为 Bool 的值,则 v 要么为 true,要么为 false。
2. 如果 v 是一个类型为 $T_1 \to T_2$ 的值,则 $v = \lambda x{:}T_1.t_2$。

证明:直接得到[类似于算术表达式的典型形式引理(8.3.1)的证明]。

利用典型形式引理,可以类似于定理(8.3.2)证明一个进展定理。这个定理的陈述需要一个小改动:我们只关心封闭项,它不含自由变量。对于开项,进展定理实际上不成立:比如项 f true 是一个范式,但不是一个值。尽管如此,这个失误不表示语言的缺陷,因为整个程序(就是那些我们实际想求值的项)总是封闭的。

9.3.5 定理[进展]:假设 t 是一个封闭的,良类型的项(即对某个 T,有$\vdash t{:}T$)。则要么 t 是一个值,要么存在某个 t′使得 $t \to t'$。

证明:直接对类型推导做归纳。布尔常量和条件式的情况与类型算术表达式进展的证明相同。变量情况不可能出现(因为 t 是封闭的)。抽象情况是直接的,因为抽象是值。

惟一有趣的情况是应用,其中 $t = t_1 t_2$ 使得$\vdash t_1 : T_{11} \to T_{12}$且$\vdash t_2 : T_{11}$。由归纳假设,要么 t_1 是一个值,要么它能做一步求值,对于 t_2 同样。如果 t_1 能做一步求值,则规则 E-App1 运用于 t。如果 t_1 是一个值并且 t_2 能做一步求值,则规则 E-App2 能运用。最后,如果 t_1 和 t_2 两者均是值,则由典型形式引理,t_1 有形式 $\lambda \vdash x{:}T_{11}.t_{12}$,且规则 E-Appabs 可运用于 t 。

下一步将要证明求值保持类型。首先陈述几个类型关系的"结构引理"。它们本身不是特别有意义,但由此可以在以后证明中对类型推导做一些有用处理。

第一个结构引理说我们可以置换一个上下文的元素,而不改变由此推导出的类型语句集合。回忆一下在一个上下文中所有的绑定必定有不同名称,并且当我们加一个囿界到一个上下文时,默认囿界的名称不同于早已囿界名称[如果需要,用约定(5.3.4)重新命名]。

9.3.6 引理[置换]:如果 $\Gamma \vdash t{:}T$ 并且 Δ 是 Γ 的一个置换,则 $\Delta \vdash t{:}T$。此外,后者的推导有着与前者推导相同的深度。

证明:直接对类型推导做归纳。

9.3.7 引理[弱化]:如果 $\Gamma \vdash t{:}T$ 并且 $x \notin dom(\Gamma)$,则 $\Gamma, x{:}S \vdash t{:}T$。此外,后者的推导有着与前者推导相同的深度。

证明:直接对类型推导做归纳。

利用这些引理,能证明类型关系的一个至关重要的性质:当用合适类型的项代换变量时,良类型性将保持。类似的引理在程序语言的安全性证明中起着普适的作用,常常称为"代换引理"。

9.3.8　引理[类型在代换下保持]:如果 $\Gamma, x:S \vdash t:T$ 并且 $\Gamma \vdash s:S$,则 $\Gamma \vdash [x \mapsto s]t:T$。

证明:对语句 $\Gamma, x:S \vdash t:T$ 的推导做归纳。对一个给定的推导,根据我们在证明中最后用到类型规则分情况讨论[①]。最有趣的情况是变量和抽象的情况。

情况 T-VAR:　$t = z$
　　　　　　　with $z:T \in (\Gamma, x:S)$

根据 z 是 x 还是另一个变量,分两个子情况考虑。如果 $z = x$,则 $[x \mapsto s]z = s$。所要的结果是 $\Gamma \vdash s:S$,这是引理的假设。否则,$[x \mapsto s]z = z$,直接得到所要的结果。

情况 T-ABS:　$t = \lambda y:T_2.t_1$
　　　　　　　$T = T_2 \to T_1$
　　　　　　　$\Gamma, x:S, y:T_2 \vdash t_1 : T_1$

由约定(5.3.4),我们假定 $x \neq y$ 并且 $y \notin FV(s)$。利用在给定的子推导中的置换,可得到 $\Gamma, y:T_2, x:S \vdash t_1:T_1$。利用在其他给定的子推导($\Gamma \vdash s:S$)中的弱化,得到 $\Gamma, y:T_2 \vdash s:S$。现在,由归纳假设,$\Gamma, y:T_2 \vdash [x \mapsto s]t_1:T_1$。由 T-Abs,$\Gamma \vdash \lambda y:T_2.[x \mapsto s]t_1:T_2 \to T_1$。但这正是我们所要的结果,因为,由代换的定义,$[x \mapsto s]t = \lambda y:T_2.[x \mapsto s]t_1$。

情况 T-APP:　$t = t_1\ t_2$
　　　　　　　$\Gamma, x:S \vdash t_1 : T_2 \to T_1$
　　　　　　　$\Gamma, x:S \vdash t_2 : T_2$
　　　　　　　$T = T_1$

由归纳假设,有 $\Gamma \vdash [x \mapsto s]t_1:T_2 \to T_1$ 和 $\Gamma \vdash [x \mapsto s]t_2:T_2$。由 T-App,有 $\Gamma \vdash [x \mapsto s]t_1\ [x \mapsto s]t_2:T$,即 $\Gamma \vdash [x \mapsto s](t_1 t_2):T$。

情况 T-TRUE:　`t = true`
　　　　　　　`T = Bool`

则 $[x \mapsto s]t = \text{true}$,并且直接得到所要的结果:$\Gamma \vdash [x \mapsto s]t:T$。

情况 T-FALSE:　`t = false`
　　　　　　　`T = Bool`

类似。

情况 T-IF:　$t = \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3$
　　　　　　$\Gamma, x:S \vdash t_1 : \text{Bool}$
　　　　　　$\Gamma, x:S \vdash t_2 : T$
　　　　　　$\Gamma, x:S \vdash t_3 : T$

---

① 或者等价地,对 t 的可能形式来分情况,因为对每个语法构造存在一个类型规则。

三次使用归纳假设得到：

$\Gamma \vdash [x \mapsto s]t_1 : \mathtt{Bool}$
$\Gamma \vdash [x \mapsto s]t_2 : \mathtt{T}$
$\Gamma \vdash [x \mapsto s]t_3 : \mathtt{T}$,

由此利用 T-If 得到结论。

利用代换引理，我们能证明类型安全性性质的另一半——求值保持良类型性。

9.3.9 定理[保持]： 如果 $\Gamma \vdash t : T$ 并且 $t \to t'$ 则 $\Gamma \vdash t' : T$。

证明： 留做练习[推荐，★★★]。除了使用代换引理，证明结构非常类似算术表达式的类型保持定理(8.3.3)的证明。

9.3.10 练习[推荐，★★]： 在练习(8.3.6)中我们研究了类型算术表达式的简单演算的主题扩展性质。这个性质是否对简单类型lambda演算的"函数部分"也成立？即假定 t 不包含任何条件表达式。$\Gamma \vdash t' : T$ 和 $t \to t'$ 是否蕴涵 $\Gamma \vdash t : T$?

9.4 Curry-Howard 对应

"→"类型构造子有两种类型规则：

1. 一个引入规则(T-Abs)，描述类型的元素是如何产生的。
2. 一个消去规则(T-App)，描述类型的元素是如何使用的。

当一个引入形式(λ)是一个消去形式(应用)的直接子项时，结果是一个约式(一个计算的机会)。

在讨论类型系统时经常使用术语"引入"和"消除"形式。当我们在本书后面接触更复杂的系统时，将看到对每个我们考虑的类型构造子有着类似的引入和消去形式。

9.4.1 练习[★]： 在图 8.1 中类型 Bool 的规则中哪些是引入规则，哪些是消除规则？图 8.2 中类型 Nat 的规则中哪些是引入规则，哪些是消除规则？

术语"引入"和"消除"来自于类型理论和逻辑之间的一个称为 Curry-Howard 对应或 Curry-Howard 同构的联系(Curry和Feys，1958；Howard，1980)。简单地说，基本思想是：在构造逻辑中，一个命题 P 的证明是由 P 的具体证据所组成的[①]。Curry 和 Howard 所注意到的是这样的证据有一个很强的计算性。比如，一个命题 $P \supset Q$ 的一个证明可以看做一个机械过程，给定 P 的证明，该过程构造 Q 的证明，或也可以这样说，Q 的证明是 P 的证明上的抽象。类似地，证明 $P \wedge Q$ 成立是要证明 P 和 Q 同时成立。根据这样的分析得出如下对应关系：

逻辑	程序语言
命题	类型
命题 $P \supset Q$	类型 $P \to Q$
命题 $P \wedge Q$	类型 $P \times Q$(参见 11.6 节)
命题 P 的证明	类型 P 的项 t
命题 P 是可证明的	类型 P 有具体内容(某项)

① 经典逻辑和构造逻辑之间的主要差别是后者的证明规则中不用排中律。这个规则是说，对每个命题 Q，要么 Q 成立，要么 $\neg Q$ 成立。在构造逻辑中为了证明 $Q \vee \neg Q$，必须提供 Q 成立的证据或 $\neg Q$ 成立的证据。

由此,简单类型 lambda 演算的一个项是对应它的类型的逻辑命题的一个证明。计算(lambda 项的归约)对应于由 cut 消除简化证明的逻辑操作。Curry-Howard 对应也称为命题比做类型。这个对应的完整讨论在许多地方都出现过,包括 Girard, Lafont 和 Taylor(1989), Gallier(1993), Sørensen 和 Urzyczyn(1998), Pfenning (2001), Goubault-Larrecq 和 Mackie (1997),以及 Simmons(2000)。

Curry-Howard 对应的优美之处在于它不限于任何特别的类型系统和相关的逻辑,相反,它能扩展为更广的类型系统和逻辑。比如,系统 F(参见第 23 章),它含有类型量词的参数多态对应于二阶构造逻辑,因为该逻辑里含有命题量词。而系统 F_ω(参见第 30 章)对应于高阶逻辑。的确,对应常常用来转换两个领域中的新进展。这样,Girard 的线性逻辑(1987)产生了线性类型系统的想法(Wadler, 1990, Wadler, 1991, Turner, Wadler 和 Mossin, 1995, Hodas, 1992, Mackie, 1994, Chirimar, Gunter 和 Riecke, 1996, Kobayashi, Pierce 和 Turner, 1996,以及许多其他文献),而模态逻辑用来帮助设计部分求值和执行时间代码生成的框架(参见 Davies 和 Pfenning, 1996, Wickline, Lee, Pfenning 和 Davies, 1998,以及他们所引用的其他文献)。

9.5 抹除和类型性

在图 9.1 中,我们直接在简单类型项上定义了求值关系。尽管类型注释在求值中不起作用(我们做任何执行时间检查来确保函数运用于合适类型的参数),在求值时确实连同考虑了这些项内部的注释。

成熟程序语言的大部分编译器实际上避免在执行时带上注释:它们在类型检查时 (以及在代码生成时和在更复杂的编译器中)使用,但在程序编译的形式中并不出现。实际上是程序在求值之前转换成一个无类型的形式。这种形式的语义能用一个抹除函数来形式化,这个抹除函数将简单类型项映射到对应的无类型项。

9.5.1 定义:一个简单类型项 t 的抹除定义为:

$$\begin{aligned} erase(\mathtt{x}) &= \mathtt{x} \\ erase(\lambda \mathtt{x}{:}\mathtt{T_1}.\ \mathtt{t_2}) &= \lambda \mathtt{x}.\ erase(\mathtt{t_2}) \\ erase(\mathtt{t_1}\ \mathtt{t_2}) &= erase(\mathtt{t_1})\ erase(\mathtt{t_2}) \end{aligned}$$

当然,我们期望简单类型演算语义的两个表示方法实际上是一致的:是直接求值一个类型项,还是先抹除类型然后求值所得到的无类型项都无关紧要。这个期望可以用下面的定理形式化,简单地说,是“求值与抹除交换”,意思是这些运算能在不同次序下进行——求值然后抹除或先抹除然后求值,得到的项相同。

9.5.2 定理:

1. 如果在类型求值关系下,$t \to t'$,那么 $erase(t) \to erase(t')$。
2. 如果在类型求值关系下,$erase(t) \to m'$,那么存在一个简单类型项 t' 使得 $t \to t'$ 并且 $erase(t') = m'$。

证明:直接对求值推导做归纳。

由于我们这里考虑的“编译”是直接的,定理(9.5.2)显然成立。对更有趣的语言和编译器,这将是一个非常重要的性质:它告诉我们一个借助于程序员写的语言直接表达的“高级”语

义与在语言的实现中实际使用的低级求值策略是一致的。

另一个抹除函数引发的有趣问题是：给定一个无类型 lambda 项 m，我们是否能找到一个简单类型项 t 使得抹除类型后成为 m?

9.5.3 定义：无类型 lambda 演算中一个项 m，如果存在某个简单类型项 t，类型 T 和上下文 Γ 使得 *erase*(t) = m 并且 $\Gamma \vdash t:T$，则 m 在 $\lambda_{\rightarrow}$ 中是可类型化的。

在第 22 章中将对这一点做深入的讨论，那时我们还将考虑与 $\lambda_{\rightarrow}$ 的类型重构紧密相关的问题。

9.6 Curry 形式和 Church 形式

我们已经看到简单类型 lambda 演算的语义形式化有两个不同的形式：作为直接定义在简单类型演算的语法上的求值关系，或作为对无类型演算和无类型项上的求值关系的编译形式。两个形式的一个重要共同点是：不管一个项 t 是否实际上是良类型的，在两个形式中均可讨论这个项 t 的行为。这种形式的语言定义称为 Curry 形式。我们首先定义项，然后定义一个语义说明它们的行为如何，然后给出一个类型系统将那些行为不合意的项删除掉。语义优先于类型。

组织一个语言定义的另一种方法是定义项，然后确定良类型项，再给出它们的语义。在这些所谓的 Church 形式的系统中，类型优先于语义：我们决不会问这样的问题："一个不良类型的项的行为是什么?"的确，严格地讲，在 Church 形式的系统中实际求值的是类型推导，而不是项(参见 15.6 节中的例子)。

历史上，lambda 演算的隐式类型表示常采用 Curry 形式，而 Church 形式的表示只用于显式类型系统。这导致了术语的混淆：Church 形式有时用于描述一个显式类型语法，而 Curry 形式有时用于描述一个隐式类型语法。

9.7 注释

Hindley 和 Seldin(1986)讨论和研究简单类型 lambda 演算，更详细的讨论可参见 Hindley (1997)。

良类型程序不会出错。
——Robin Milner (1978)

第 10 章　简单类型的 ML 实现[①]

作为 ML 程序的 $\lambda_{\to}$ 的具体实现与第 7 章中无类型 lambda 演算的实现相同。主要不同之处是一个函数 typeof,它计算一个给定项在给定上下文中的类型。在此之前,我们先提出一个低层机理来处理上下文。

10.1　上下文

回忆一下第 7 章,上下文就是由变量名和 binding 组成的序对的列表:

```
type context = (string * binding) list
```

在第 7 章中,上下文只用于在分析和打印期间在项的命名和无命名的形式之间的转换。为此,我们只需要知道变量的名称;binding 为一个不带任何信息的构造子的数据类型:

```
type binding = NameBind
```

为了实现类型检查器,将需要用上下文携带关于变量的类型假设。通过在 binding 类型上加上一个称为 VarBind 的新构造子来实现这一点:

```
type binding =
    NameBind
  | VarBind of ty
```

每个 VarBind 构造子携带相应变量的类型假设。为了方便不关心类型假设的打印和分析的函数,除了 VarBind 外,保持老的 NameBind 构造子(实现策略是定义两个完全不同 context 类型——一个用来分析和打印,另一个用来类型检查)。

函数 typeof 利用一个函数 addbinding 来扩展一个含新变量绑定(x,bind)的上下文 ctx;因为上下文表示为列表,addbinding 本质上就是 cons:

```
let addbinding ctx x bind = (x,bind)::ctx
```

反之,我们用函数 getTypeFromContext 在一个上下文 ctx 中抽取与一个特殊变量 i 有关的类型假设(如果 i 超出范围,文件信息 fi 会打印一个错误消息):

```
let getTypeFromContext fi ctx i =
  match getbinding fi ctx i with
      VarBind(tyT) → tyT
    | _ → error fi
      ("getTypeFromContext: Wrong kind of binding for variable "
       ^ (index2name fi ctx i))
```

match 提供某种内部一致性检查:在规范环境下, getTypeFromContext 总是在一个上下文中调用,在这个上下文中,第 i 个绑定事实上是一个 VarBind。在后面的章节中,我们将加入其他形

① 这里描述的实现对应于带布尔类型(参见图 8.1)的简单类型 lambda 演算(参见图 9.1)。本章中的代码可以在 Web 库的 simplebool 中找到。

式的绑定(特别是对类型变量的绑定),并可能用错误变量调用 getTypeFromContext。在这种情况下,用低级 error 函数来打印一个消息,将消息传给 info 使得它能报告出现错误的文件位置。

```
val error : info → string → 'a
```

error 函数的结果类型是变量类型'a,它能被实例化为任何 ML 类型(这是有意义的,因为它不会返回:它打印一个消息然后停止程序)。这里,我们需要假设 error 的结果是一个 ty,因为这是 match 的另一分支所返回的。

注意我们用索引来搜索类型假设,因为项内部表示为无命名形式,而变量表示为数值索引。函数 getbinding 在给定的上下文中简单搜索第 i 个绑定:

```
val getbinding : info → context → int → binding
```

它的定义可以在本书网站上的 simplebool 实现中找到。

10.2 项和类型

类型的语法可直接从图 8.1 和图 9.1 中的抽象语法转换为一个 ML 数据类型。

```
type ty =
    TyBool
  | TyArr of ty * ty
```

项的表示如同我们在无类型 lambda 演算中一样,只是加入一个类型注释到 TmAbs 子句中:

```
type term =
    TmTrue of info
  | TmFalse of info
  | TmIf of info * term * term * term
  | TmVar of info * int * int
  | TmAbs of info * string * ty * term
  | TmApp of info * term * term
```

10.3 类型检查

类型检查函数 typeof 可以看做是 $\lambda_{\rightarrow}$ 的类型规则(参见图 8.1 和图 9.1)的一个直接翻译,或更精确地,是逆转引理(9.3.1)的一个副本。第二个观点是更加精确的,因为正是逆转引理告诉我们,对每个语法形式,这个形式的一个项是良类型必须满足什么条件。类型规则告诉我们一定形式的项在某些条件下是良类型的,但考虑到个别的类型规则,就不能得出这样的结论:某个项不是良类型的,因为总有可能存在别的规则来类型化该项(此时,可以出现不明显的差别,因为逆转引理是直接从类型规则得出的。但在以后的系统中证明逆转引理比在 $\lambda_{\rightarrow}$ 中证明更复杂,所以这个差别会变得更加重要)。

```
let rec typeof ctx t =
  match t with
    TmTrue(fi) →
      TyBool
```

```
| TmFalse(fi) ->
    TyBool
| TmIf(fi,t1,t2,t3) ->
   if (=) (typeof ctx t1) TyBool then
     let tyT2 = typeof ctx t2 in
     if (=) tyT2 (typeof ctx t3) then tyT2
     else error fi "arms of conditional have different types"
   else error fi "guard of conditional not a boolean"
| TmVar(fi,i,_) -> getTypeFromContext fi ctx i
| TmAbs(fi,x,tyT1,t2) ->
    let ctx' = addbinding ctx x (VarBind(tyT1)) in
    let tyT2 = typeof ctx' t2 in
    TyArr(tyT1, tyT2)
| TmApp(fi,t1,t2) ->
    let tyT1 = typeof ctx t1 in
    let tyT2 = typeof ctx t2 in
    (match tyT1 with
        TyArr(tyT11,tyT12) ->
          if (=) tyT2 tyT11 then tyT12
          else error fi "parameter type mismatch"
      | _ -> error fi "arrow type expected")
```

这里,值得指出 OCaml 语言的几个细节。首先,OCaml 相等算子 = 写在括号内,因为我们在前缀位置中用它,而不是在正规的中缀位置来帮助与后面出现的 typeof 做比较,那时的比较类型操作将比简单相等更为细致。第二,相等算子计算复合值上的一个结构相等,不是一个指针相等。即表达式:

```
let t  = TmApp(t1,t2) in
let t' = TmApp(t1,t2) in
(=) t t'
```

保证产生 true,尽管 TmApp 囿界 t,t'的两个实例在不同时间分配内存并且存储在不同地址上。

第11章　简单扩展[①]

简单类型 lambda 演算有着足够丰富的结构来讨论它的理论性质,但对于一个程序语言来说仍然不够。在这一章中,开始引入几个可以直接处理类型化的,熟悉的特征,以使我们所讨论的更接近大家熟悉的语言。出现在本章中的一个重要概念是导出形式。

11.1　基本类型

每个程序语言提供各种基本类型(简单的无结构值的集合,如数值,布尔值和字符)加上对这些值处理的合适的原语操作。我们早已详细地讨论了自然数和布尔值;语言设计者所希望的许多其他基本类型也能用同样的方式加进来。

除了 Bool 和 Nat ,我们将用基本类型 String(如 hello 这样的元素)和 Float(如 3.14159 这样的元素)来丰富本书中例子。

从理论的角度出发,通常抽取出特殊基本类型和它们的运算细节,而简单地假定我们的语言含有一个具有无解释或未知的基类型,且没有原语操作的集合$\mathcal{A}$。这可以通过简单地将$\mathcal{A}$的元素(元变量 A)加到类型集合,如图 11.1 所示就可以做到这一点。我们用字母$\mathcal{A}$而不是字母$\mathcal{B}$表示基本类型,来避免和$\mathbb{B}$符号的混淆,这个符号已用来表示一个给定的系统中布尔类型。$\mathcal{A}$能看做是表示原子类型(基本类型的另一个名称),因为就类型系统而言它们没有内部结构。我们将用 A,B,C 等作为基本类型的名称。注意,如我们处理变量和类型变量一样,用 A 表示基本类型,同时也表示基本类型上的元变量,应从上下文来判断它究竟是基本类型还是基本类型的元变量。

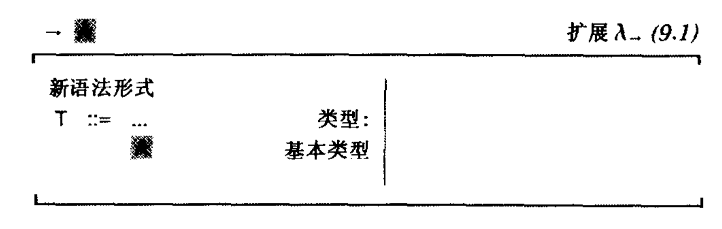


图 11.1　无解释的基本类型

一个没有解释的类型有用吗?答案是肯定的。尽管我们没有办法直接命名它的元素,但仍能绑定取值于一个基本类型上的变量。比如,函数[②]:

```
λx:A. x;
▸ <fun> : A → A
```

是 A 的元素上的恒等函数,不管这些元素是什么。同样:

---

① 本章中讨论的系统是纯类型 lambda 演算(参见图 9.1)的各种扩展。相应的 OCaml 实现,fullsimple 包括所有这些扩展。

② 从现在起,当我们展示求值的结果时,将省略 λ 抽象体——将它们只记为 <fun>。

```
λx:B. x;
  ▸ <fun> : B → B
```

是 B 上的恒等函数,而:

```
λf:A→A. λx:A. f(f(x));
  ▸ <fun> : (A→A) → A → A
```

是一个函数,它重复两次函数 f 在一个参数 x 上的行为。

11.2　单位类型

另一个有用的基本类型,特别是应用在 ML 系列的语言中,是图 11.2 中描述的单个类型(Unit)。与上一节中无解释的基本类型相比,这个类型是可以用最可能简单的方式进行解释的:我们显式地引入一个单个元素[项常量unit(记为小写的 u)]和一个类型规则使unit是Unit的一个元素。我们也将unit加到计算可能结果的值的集合中去。的确,unit是求值类型Unit的表达式产生的惟一的可能结果。

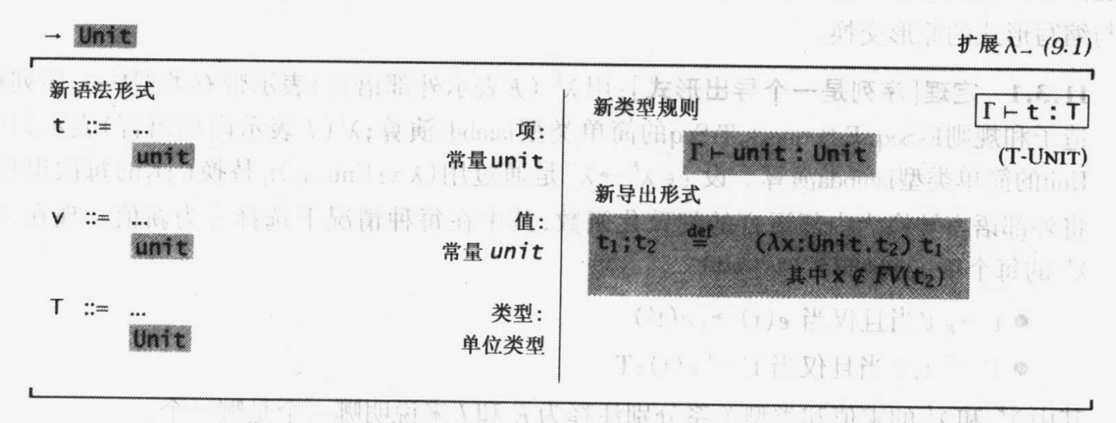


图 11.2　单位类型

即使是在一个纯函数式的语言中,类型 Unit 也不是完全没有意义的[①],但它的主要应用是在带副作用的语言中,如给引用单元赋值(在第 13 章将讨论这个话题)。在这样的语言中,我们所关心的往往是表达式的副作用,而不是结果;Unit 是这样表达式的一个合适的结果型。

Unit 的作用类似于 C 和 Java 这样的语言中的 void 类型。虽然名称 void 类似于空类型 Bot(参见 15.4 节),但 void 的使用实际上更接近我们的 Unit。

11.3　导出形式:序列和通配符

在有副作用的语言中,对序列中的两个或多个表达式求值常常是有用的。序列记号 $t_1;t_2$ 有这样的效果:求值 t_1,扔掉它平凡的结果,然后继续求值 t_2。

① 读者可能喜欢下面的问题:11.2.1 练习[★★★]:在只带基本类型 Unit 的简单类型 lambda 演算中是否为有一种方法构造一个项的序列 $t_1, t_2, \cdots$,使得对每个 n,项 t_n 的长度至多是 $O(n)$,但求值到一个范式所要求的步骤至少为 $O(2^n)$?

实际上有两个不同的方法来形式化序列。一个是采用对其他语法形式所采用的方式:在项的语法中加入一个新的定义“$t_1;t_2$”和两个求值规则:

$$\frac{t_1 \longrightarrow t_1'}{t_1;t_2 \longrightarrow t_1';t_2} \qquad \text{(E-Seq)}$$

$$\texttt{unit};t_2 \longrightarrow t_2$$

以及一个类型规则:

$$\frac{\Gamma \vdash t_1 : \texttt{Unit} \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1;t_2 : T_2} \qquad \text{(T-Seq)}$$

来捕捉“;”的行为。

形式化序列的另一个方法是简单地将 $t_1;t_2$ 看做是项$(\lambda x\text{:Unit}.t_2)\ t_1$ 的一个缩写,其中变量 x 选择为新变量,即不同于 t_2 中的所有的自由变量。

直观上很清楚,对程序员而言,这两种序列的表示是相同的:序列的高级类型化和求值规则能由$(\lambda x\text{:Unit}.t_2)\ t_1$ 的缩写形式 $t_1;t_2$ 导出。这个直观对应更形式地表示为:类型化和求值都能与缩写形式的原形交换。

11.3.1 定理[序列是一个导出形式]:用 λ^E (E表示外部语言)表示带有类型Unit,序列构造子和规则E-Seq,E-Seqnext,T-Seq的简单类型lambda演算;λ^I (I 表示内部语言)表示只带Unit的简单类型lambda演算。设 $e \in \lambda^E \to \lambda^I$ 是通过用$(\lambda x\text{:Unit}.t_2)\ t_1$ 替换$t_1;t_2$的每次出现,将外部语言转换为内部语言的细致化函数,其中在每种情况下选择 x 为新值。现在,对 λ^E 的每个项 t,我们有:

- $t \to_E t'$ 当且仅当 $e(t) \to_I e(t')$
- $\Gamma \vdash^E t\text{:}T$ 当且仅当 $\Gamma \vdash^I e(t)\text{:}T$

其中 λ^E 和 λ^I 的求值和类型关系分别注释为 E 和 I 来说明哪一个是哪一个。

证明:“当且仅当”的两个方向的证明可以直接对 t 的结构做归纳证明。

定理(11.3.1)说明了术语导出形式的使用合法性,因为它说明序列构造子的类型和求值行为能由抽象和应用更基本操作的类型和求值行为推导出。这样引入序列作为导出形式,而不是作为成熟语言构造的好处是能扩展表面语法(即程序员实际用来写程序的语言)而不增加内部语言的复杂性,而且这些内部语言中类型安全性等定理必须证明。这种分解语言特征描述的方法早已用在Algol 60的报告(Naur等,1963)中,并且大量用于许多新近的语言定义中,如标准ML的定义(Milner,Tofte和Harper,1990; Milner,Tofte,Harper和MacQueen,1997)。

根据Landin,导出形式常常称为语法修饰,用它的低级定义替代一个导出形式称为化简。

将在后面例子中用到的另一个导出形式是变量绑定器的“通配符”约定。比如,在化简序列后产生的项中,我们常常写一个“虚拟”lambda抽象,其中参数变量不是真正用在抽象体中。在这种情况下,常常不得不明显地选择一个囿变量的名称;而我们更喜欢用一个通配符绑定器来替代它,记为_。即用 λ_:S.t 表示 λx:S.t 的缩写,其中 x 是某个在 t 中不出现的变量。

11.3.2 练习[★]:给出通配符抽象的类型和求值规则,并且证明它们能从上述的缩写中推导出。

11.4　归属

另一个常用的简单特征是能够明显地将一个特别类型归属为一个给定的项(即在程序的文本中记录这个项有这个类型)。用"t as T"表示"我们将类型 T 归属到项 t"。这个构造子(参见图 11.3)的类型规则 T-Ascribe 简单地验证了所归属的类型 T 的确是 t 的类型。求值规则 E-Ascribe 也是直接的:它只是丢掉归属,让 t 自由地求值。

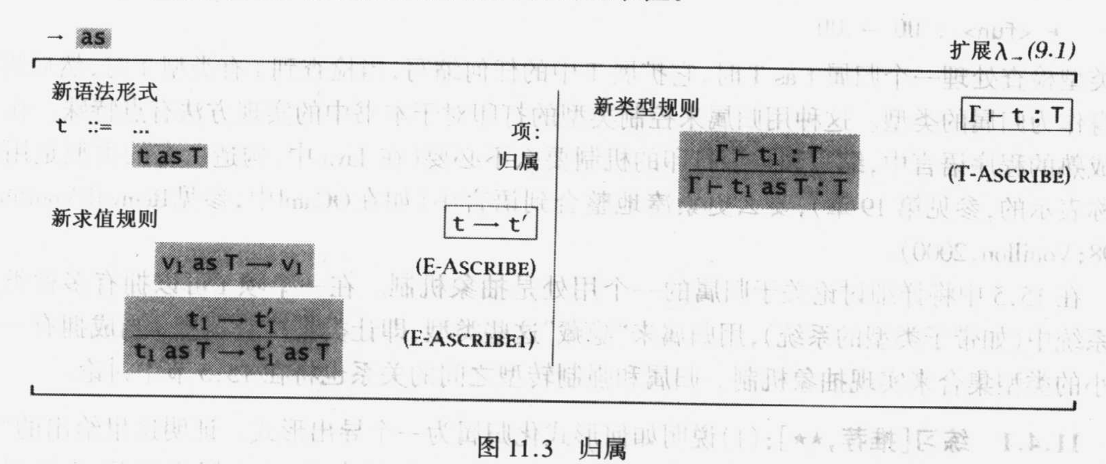


图 11.3　归属

在编程时有几种情况能用到归属。一个常用的是文档。它有时使读者难以得到一个大的复合表达式的子表达式类型。合法地使用归属可以使程序易读。类似地,在一个特别复杂的表达式中,甚至连程序员也不是很清楚所有的子表达式的类型是什么。加入若干归属是一个很好的办法来清晰程序员的思路。的确,归属有时在查明令人不解的类型错误的来源时是有帮助的。

归属的另一个用处是控制复杂类型的打印。本书中用来检查例子的类型检查器——及其 OCaml 实现(它的名称带上前缀 full)提供了一个简单的机制来引入长的或复杂类型表达式的缩写(为了易于阅读和修改常常在其他实现中省略缩写机制)。比如,声明:

```
UU = Unit→Unit;
```

用 UU 表示 Unit→Unit 的一个缩写。当见到 UU 时,就能理解为 Unit→Unit。比如,我们能写:

```
(λf:UU. f unit) (λx:Unit. x);
```

在类型检查的过程中,这些缩写必要时可以自动扩张。相反地,类型检查器则尽可能地采用缩写形式(特别地,每次它们计算一个子项的类型时,检查是否这个项正好匹配任何当前定义的缩写,如果匹配,就用缩写替代类型)。这一般会给出合理的结果,但有时我们可以要求一个类型有不同的打印方式,这是因为简单的匹配策略导致类型检查器忽略采用一个缩写的机会(比如,在记录类型的字段可以置换的系统中,它将不知道{a:Bool ,b:Nat}是与{b:Nat ,a:Bool}可交换的),或者是因为其他的原因。比如,在:

```
λf:Unit→Unit. f;
▸ <fun> : (Unit→Unit) → UU
```

缩写 UU 是在函数的结果中压缩形式,但在它的参数中不压缩。如果我们要求类型打印为 UU →UU,可以要么改变抽象上的类型注释:

```
λf:UU. f;
▸ <fun> : UU → UU
```

要么在整个抽象中加入一个归属:

```
(λf:Unit→Unit. f) as UU→UU;

▸ <fun> : UU → UU
```

当类型检查处理一个归属 t as T 时,它扩展 T 中的任何缩写,当检查到 t 有类型 T 时,然后将 T 本身作为归属的类型。这种用归属来控制类型的打印对于本书中的实现方法有点特殊。在一个成熟的程序语言中,缩写和类型打印的机制要么不必要(在 Java 中,构造所有的类型是用短名称表示的,参见第 19 章),要么更紧凑地整合到语言中(如在OCaml中,参见Rémy和Vouillon,1998;Vouillon,2000)。

在 15.5 中将详细讨论关于归属的一个用处是抽象机制。在一个项 t 可以拥有多种类型的系统中(如带子类型的系统),用归属来"隐藏"这些类型,即让类型检查器把 t 当成拥有一个更小的类型集合来实现抽象机制。归属和强制转型之间的关系也将在 15.5 节中讨论。

11.4.1 练习[推荐,★★]:(1)说明如何形式化归属为一个导出形式。证明这里给出的"正式"类型和求值规则在一个适当的意义下对应于你的定义;(2)假定不用求值规则 E-Ascribe和E-Ascribe1,我们给出一个"迫切"规则:

$$t_1 \text{ as } T \longrightarrow t_1 \qquad \text{(E-AscribeEager)}$$

尽早丢掉归属。归属是否仍能看做是一个导出形式?

11.5 let 绑定

当写一个复杂表达式时(为了避免重复和增加可读性),常常给出某些子表达式名称。大部分语言提供一个和多个办法来做到这一点。比如,在 ML 中,我们写let x $=t_1$ in t_2来表示求值表达式 t_1,并且在求值 t_2 时绑定名称 x 为结果值。

我们的 let 绑定器(如图 11.4 所示)是根据 ML 的按值调用的求值顺序,其中 let 囿项必须在对 let 体的求值之前完全求值。类型规则 T-Let 告诉我们一个 let 的类型可以这样来计算:通过计算 let 囿项的类型,扩展带有这个类型的一个绑定上下文,并且在这个扩展的上下文中计算体的类型,这样得出的结果就是整个 let 表达式的类型。

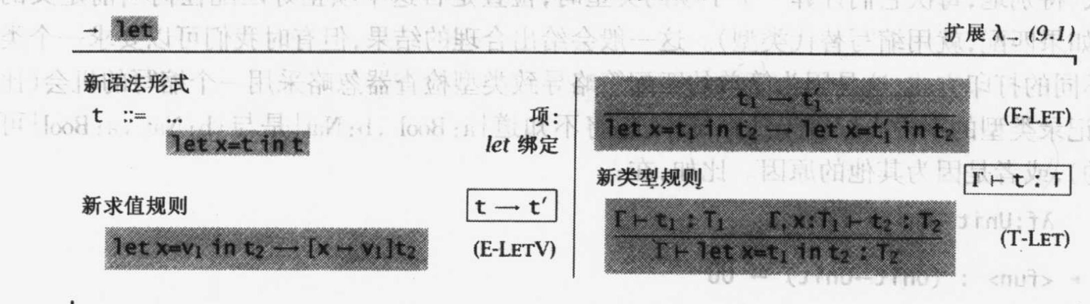


图 11.4 let 绑定

11.5.1 练习[推荐,★★★]:letexercise类型检查器(在本书的网站上可以查到)是一个 let 表达式的不完全实现:提供了基本的分析和打印函数,但在 eval1 和 typeof 函数中漏掉了 TmLet的子句(在它们的位置,会出现与所有都匹配并使程序失败的虚拟子句,请将 TmLet 子句补上。

let 也能定义为一个导出形式吗?是的,如 Landin 所证明的;但细节比我们对序列和归属做的方法要复杂。直观地,很清楚我们可以用抽象和应用的组合来达到一个 let 绑定的效果:

$$\texttt{let } x\texttt{=}t_1 \texttt{ in } t_2 \quad \stackrel{\text{def}}{=} \quad (\lambda x\texttt{:}T_1\texttt{.}t_2)\ t_1$$

但注意到这个缩写的右端包括类型注释 T_1,它不出现在左端。即,如果我们将导出式想像为在某个编译器的语法分析阶段的化简,则需要问如何假设分析器知道它应该产生的 T_1,是作为在化简后的内部语言项中对 λ 的类型注释。

当然,回答是这个信息来自类型检查器。通过计算 t_1 的类型找到所需的类型注释。更形式地,这告诉我们构造子 let 是不同与我们到现在为止所见的导出形式:我们不应该将它看做是一个项上化简后的转换,而应该将它看做是类型推导的一个转换(或是对带有化简后分析结果的类型检查器的项的转换),这个转换将含有一个 let 推导:

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash t_1 : T_1} \qquad \dfrac{\vdots}{\Gamma, x\texttt{:}T_1 \vdash t_2 : T_2}}{\Gamma \vdash \texttt{let } x\texttt{=}t_1 \texttt{ in } t_2 : T_2}\ \text{T-LET}$$

映射为用抽象和应用的一个推导:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, x\texttt{:}T_1 \vdash t_2 : T_2}}{\Gamma \vdash \lambda x\texttt{:}T_1\texttt{.}t_2 : T_1 \to T_2}\ \text{T-ABS} \qquad \dfrac{\vdots}{\Gamma \vdash t_1 : T_1}}{\Gamma \vdash (\lambda x\texttt{:}T_1\texttt{.}t_2)\ t_1 : T_2}\ \text{T-APP}$$

这样,比起我们见到的其他的导出形式,let 有些不同:我们能通过化简来推导它的求值行为,但它的类型行为必须建立在内部语言中。

在第 22 章中,将看到另一个原因使我们不将 let 处理为一个导出形式:在带 Hindley-Milner 的(即基于合一的)多态的语言中,类型检查器特别处理 let 构造子,用这个构造子来推广多态定义以得到用普通 λ 抽象和应用不能仿效的类型。

11.5.2 练习[★★]:定义 let 为一个导出形式的另一种方法可以通过直接"执行"它,将它化简,即将 let $x = t_1$ in t_2 看做是代换体$[x \mapsto t_1]t_2$ 的缩写。这是一个好想法吗?

11.6 序对

大部分程序语言提供各种方法来建立复合数据结构。其中最简单的是序对,或更一般的说法是值的元组。在本节中处理序对,然后在 11.7 节和 11.8 节中讨论元组和标签记录的更一般形式①。

① fullsimple 的实现,实际上没有提供这里序对的语法,因为元组是更一般的形式。

序对的形式化太简单了，无需讨论。到现在为止我们能像读一段文字描述那样很容易理解图 11.5 中的规则。尽管如此，我们还是简单地看一下以普通模式定义的各个部分。

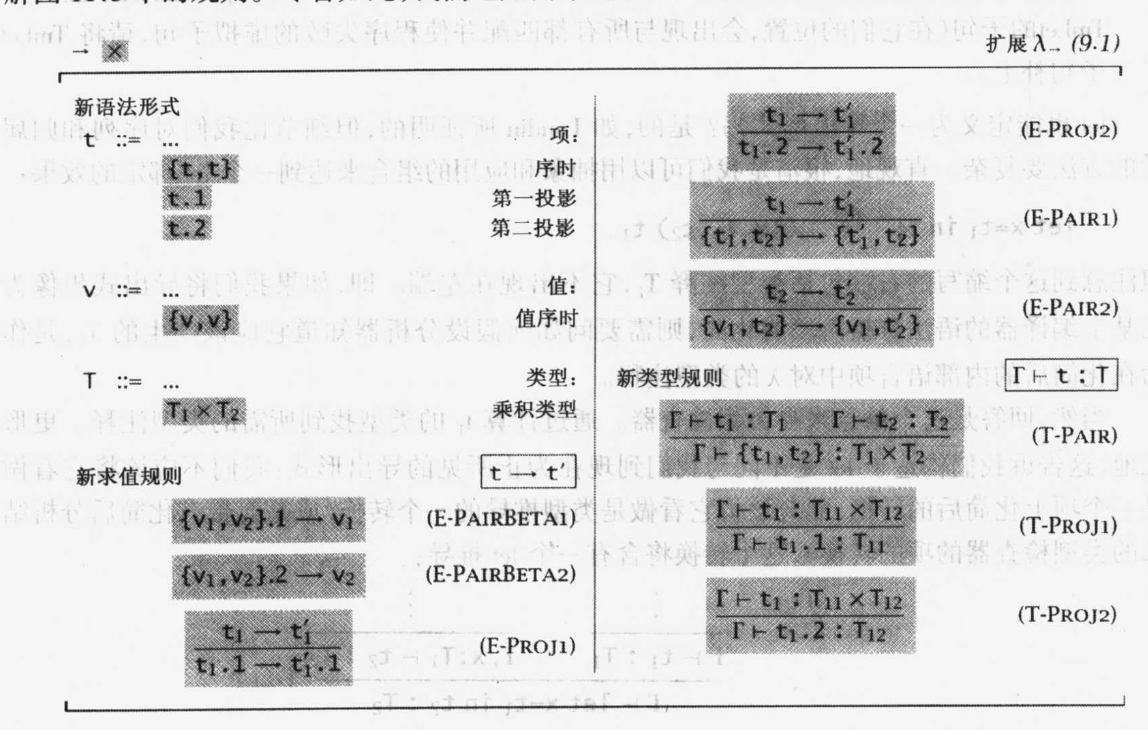


图 11.5 序对

将序对加入到简单类型 lambda 演算包括将项的两个新形式的项 – 序对，记为 $\{t_1, t_2\}$ 和投影，t 第一投影记为 t.1，t.2 表示 t 的第二投影，加上一个新的类型构造子 $T_1 \times T_2$，称为 T_1 和 T_2 的乘积(或有时称为笛卡儿乘积)。用大括号表示序对①，以强调与 11.8 节中记录的联系。

对于求值，我们需要几个新的规则说明序对和投影的行为。E-PairBeta1 和 E-PairBeta2 说明当一个完全求值的序对遇到一个第一或第二投影时，结果是适当的分量。当被投影的项还没有完全求值时，E-Proj1 和 E-Proj2 允许在投影下归约。E-Pair1 和 E-Pair2 对序对中的分量求值：首先是左边的分量，然后是当左边变成一个值时再对右边分量求值。

在这些规则，中元变量 v 和 t 的使用次序使我们必须采用从左到右的求值策略。比如，组合项：

```
{pred 4, if true then false else false}.1
```

只能求值如下：

```
   {pred 4, if true then false else false}.1
→  {3, if true then false else false}.1
→  {3, false}.1
→  3
```

---

① 大括号对序对和元组不合适，因为大括号是数学中表示集合的符号。在像 ML 这样的大众语言中和研究文献中通常将序对和元组括在小括号中。也采用其他像方括号或尖括号的符号。

我们也需要加入一个新的子句到值的定义中，以说明 $\{v_1, v_2\}$ 是一个值。一个序对的值的分量必须本身也是值，这样可保证将序对作为参数传递给函数时已经经过了完全求值以供函数体执行时用。比如：

```
   (λx:Nat×Nat. x.2) {pred 4, pred 5}
→  (λx:Nat×Nat. x.2) {3, pred 5}
→  (λx:Nat×Nat. x.2) {3,4}
→  {3,4}.2
→  4
```

序对和投影的类型规则是直接的。引入规则 T-Pair 说明：如果 t_1 有类型 T_1 并且 t_2 有类型 T_2，则 $\{t_1, t_2\}$ 有类型 $T_1 \times T_2$。反之，消规则 T-Proj1 和 T-Proj2 告诉我们：如果 t_1 有一个乘积类型 $T_{11} \times T_{12}$（即如果它将求值为一个序对），则这个序对投影的类型是 T_{11} 和 T_{12}。

11.7　元组

很容易推广上一节中二元乘积到 n 元乘积（常称为元组）。比如{1,2,true}是一个三元组，包含两个数和一个布尔值。它的类型记为{Nat ,Nat ,Bool}。

这个推广的惟一代价是为形式化这个系统，我们需要提出一些符号来统一地描述任意元的结构；这样的符号总是存在问题的，因为在严密性和可读性之间总有不可避免的矛盾。我们用 $\{t_i{}^{i\in 1..n}\}$ 表示 n 个项 $t_1, \cdots, t_n$ 的元组，并且用 $\{T_i{}^{i\in 1..n}\}$ 表示它的类型。注意这里的 n 可认为 0；在这种情况下，范围 $1\cdots n$ 为空，并且 $\{t_i{}^{i\in 1..n}\}$ 是{}，即空元组。注意值 5 与含一个元素的元组{5}是有差别的：我们对后者惟一合法的操作是可以投影它的第一分量。

图 11.6 形式化了元组。定义类似于乘积（如图 11.5 所示）的定义，只是序对的每个规则推广到 n 元情况，且对第一和第二投影的每对规则变成对元组做任意投影的单个规则。惟一需要说明的规则是 E-Tuple，它结合并推广了图 11.5 中的规则 E-Pair1 和 E-Pair2。也就是说，如果我们有一个元组中所有字段 j 左边的字段已经归约为值，则这个字段能从 t_j 一步求值到 t'_j。元变量的使用使我们采用从左到右的求值策略。

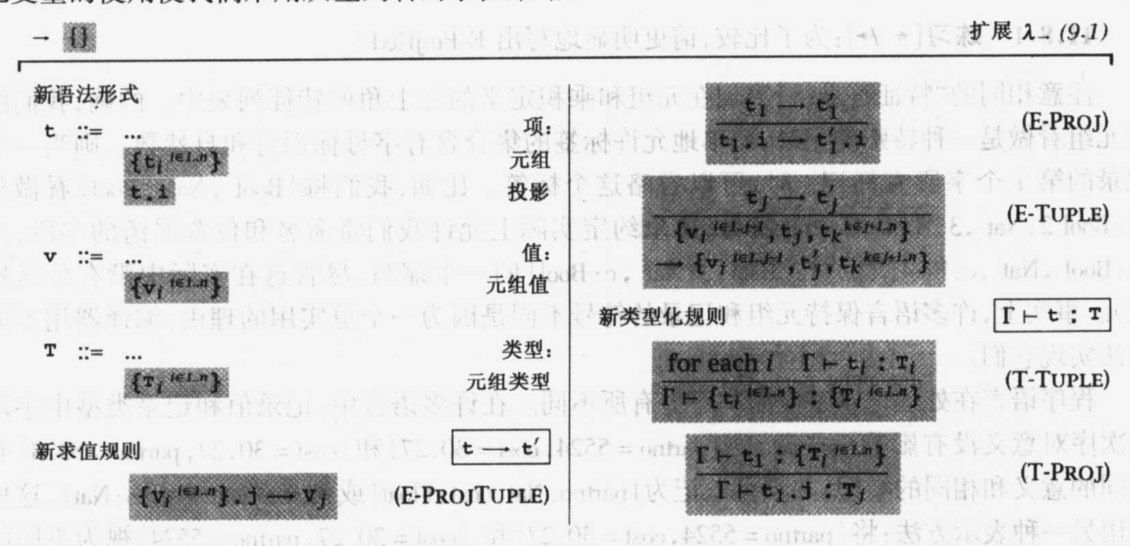


图 11.6　元组

11.8 记录

从 n 元组推广到带标签的记录也是直接的。对每个字段 t_i，我们简单从某个事先确定的集合$\mathcal{L}$中取一个标签 l_i 注释字段 t_i。比如，{x = 5}和{partno = 5524, cost = 30.27}是记录值；它们的类型是{x: Nat}和{partno: Nat , cost: Float}。要求在一个给定的记录项或类型中所有的标签是不同的。

图 11.7 给出记录的规则。惟一值得注意的规则是 E-ProjRcd，它依赖一个稍微非形式的约定。这个规则应理解如下：如果$\{l_i = v_i{}^{i\in 1..n}\}$是一个记录，并且 l_j 是它的第j 个字段的一个标签，则$\{l_i = v_i{}^{i\in 1..n}\}.l_j$ 一步求值为第j 个值 v_j。这个约定（和在 E-ProjTuple 中用的类似）能通过用一个更明显的形式重写规则来消除它；尽管如此，可读性的代价是相当高的。

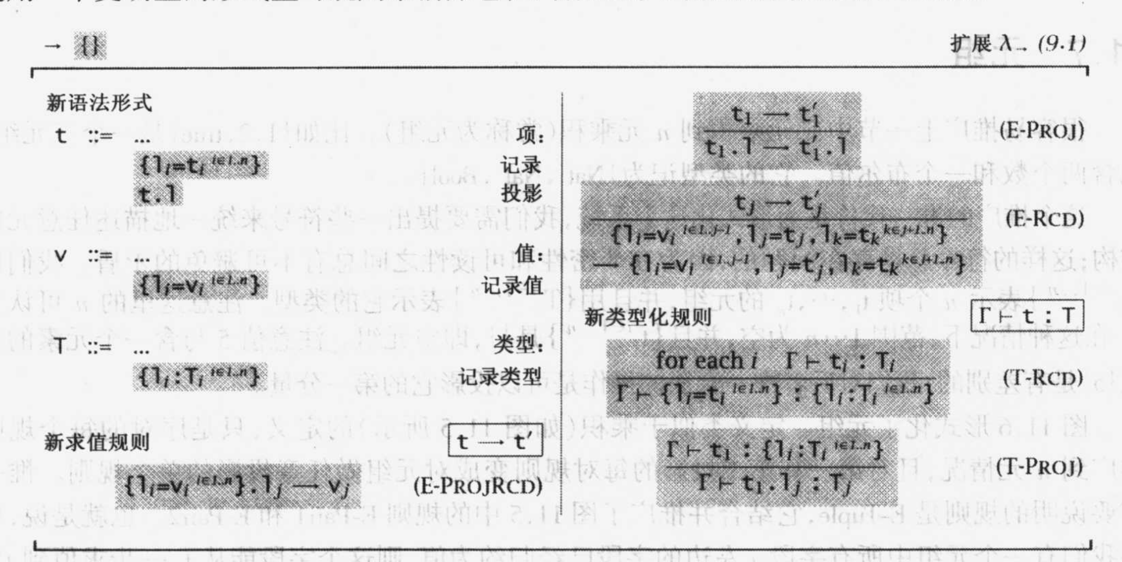


图 11.7 记录

11.8.1 练习[★ ↛]：为了比较，请更明显地写出 E-ProjRcd。

注意相同的"特征符号{}"出现在元组和乘积定义的左上角的特征列表中。的确，我们能将元组看做是一种特殊的记录，简单地允许标签的集合含有字母标识符和自然数。则当一个记录的第 i 个字段有标记 i 时，可以省略这个标签。比如，我们将{Bool , Nat , Bool}看做是{1: Bool 2: Nat , 3: Bool}的一个缩写（这个约定实际上允许我们将命名和位置混淆的字段，用{a: Bool , Nat , c: Bool}表示{a: Bool, 2: Nat , c: Bool}的一个缩写，尽管这在实际中没有什么用处）。事实上，许多语言保持元组和记录的符号不同是因为一个更实用的理由：编译器用不同方法实现它们。

程序语言在处理记录字段的次序上有所不同。在许多语言中，记录值和记录类型中字段的次序对意义没有影响——即，项{partno = 5524, cost = 30.27}和{cost = 30.27, partno = 5524}有相同的意义和相同的类型，并且可以记为{partno: Nat, cost: Float}或{cost: Float, partno: Nat}。这里采用另一种表示方法：将{partno = 5524, cost = 30.27}和 {cost = 30.27, partno = 5524}视为不同的记录值，类型分别是{partno: Nat, cost: Float}和{cost: Float, partno: Nat}。在第 15 章中，对次序将采

用一个更自由的观点，引入一个子类型关系，其中类型{partno: Nat, cost: Float}和{cost: Float, partno: Nat}是等价的(每个是另一个的子类型)，使得一个类型的项可以用在任何一个期望得到其他类型的上下文中(有了子类型后，有序和无序的记录之间的选择对性能有着重要的影响；这些将在 15.6 节中进一步讨论。一旦确定采用无序的记录，选择是否考虑记录在开始时是无序的，还是原始地将字段看做是有序的，然后给出可以忽略序的规则，这是一个看法的问题。这里采用后一种处理，因为它允许我们讨论这两个选择)。

11.8.2 练习[★★★]：在我们记录的表示中，投影操作用来一次抽取记录的一个字段。许多高级程序语言提供了另一种模式匹配语法，即一次抽取所有的字段，让程序可以简洁地表示。模式也可以是嵌套的，允许很容易地从复杂的嵌套的数据结构中抽取出一部分来。我们能在带记录的无类型 lambda 演算上加入一个简单形式的模式匹配，通过加一个新模式语法范畴，以及一个(模式匹配构造子本身)新的情况到项的语法中(参见图 11.8)。

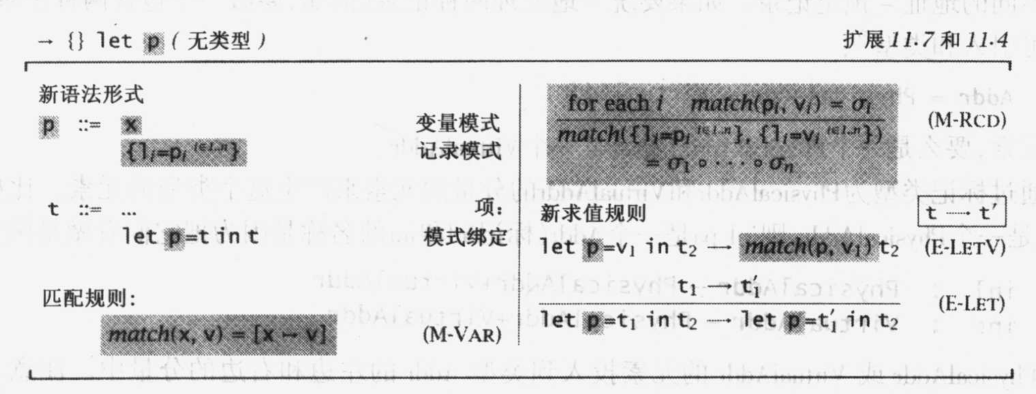


图 11.8 (无类型的)记录模式

用模式匹配的计算规则来推广图 11.4 中的 let 绑定规则。它依赖一个辅助的"匹配"函数：给定一个模式 p 和一个值 v，这个函数要么失败(指出 v 与 p 不匹配)，要么产生一个代换，将出现在 p 中变量映射到 v 的相应部分。比如，*match*({x,y}，{5,true})产生代换[x ↦ 5, y ↦ true]，*match*(x，{5,true})产生[x ↦ {5,true}]，而 *match*({x}，{5,true})失败。E-LetV 使用 *match* 来给 p 中的变量计算一个的合适代换。

match 函数本身是由一个推论规则集合所定义的。M-Var 规则说明一个变量模式总是成功地送回一个，将变量映射到与之匹配的整个值的代换。M-Rcd 规则说明为了匹配一个记录模式$\{l_i = p_i{}^{i\in 1..n}\}$与一个(相同长度的，带相同标签的)记录值$\{l_i = v_i{}^{i\in 1..n}\}$，我们分别匹配每个子模式 p_i 与相应的值 v_i 来得到一个代换 σ_i，并通过合成所有这些代换以建立最终的代换(要求在一个模式中没有变量出现多于一次，这样代换的合成就是它们的并)。下面请说明如何在这个系统中加入类型。

1. 给出新构造子的类型规则(必要时，可对语法做修改)。
2. 写出整个演算的类型保持和进展定理的一个证明提纲(不需要给出整个证明，只是用正确的次序陈述所需要的引理)。

11.9 和

许多程序需要处理值的异类集合。比如,一个二叉树上的一个节点可以是一个叶子,也可以是一个带有两个子节点的内部节点;类似地,一个列表单元可以是 nil 或是一个带有一个头和一个尾的 cons 单元[①],在编译器中一个抽象语法树的一个节点能表示一个变量、一个抽象和一个应用等。支持这种程序的类型论机制是变式类型。

在完全一般化引入变式之前(参见 11.10 节),考虑一个较简单的情况:二元和类型。一个和类型描述从两个给定类型中取值的集合。比如,假定我们用类型:

```
PhysicalAddr = {firstlast:String, addr:String};
VirtualAddr  = {name:String, email:String};
```

表示不同的地址 - 预定记录。如果要统一地处理两种记录(例如,要做一个包含两种记录的列表),可引入和类型[②]:

```
Addr = PhysicalAddr + VirtualAddr;
```

它的元素,要么是一个PhysicalAddr,要么是一个VirtualAddr。

通过标记类型为PhysicalAddr和VirtualAddr的分量的元素来产生这个类型的元素。比如,如果 pa 是一个PhysicalAddr,则inl pa是一个Addr(标记inl和inr的名称是因为把它们看做是函数:

```
inl  :  PhysicalAddr → PhysicalAddr+VirtualAddr
inr  :  VirtualAddr → PhysicalAddr+VirtualAddr
```

即将 PhysicalAddr 或 VirtualAddr 的元素投入到类型 Addr 的左边和右边的分量中。注意,尽管如此,它们在我们的表示中不作为函数处理)。

一般地,一个类型 T_1+T_2 的元素是由标记为 inl 的 T_1 元素和标记为 inr 的 T_2 元素组成的。

为了使用和类型的元素,引入一个构造子 case,允许我们区分一个值是来自和类型左边的还是右边的分支。比如,我们能从一个 Addr 中抽取一个名称,像:

```
getName = λa:Addr.
  case a of
    inl x ⇒ x.firstlast
  | inr y ⇒ y.name;
```

当参数 a 是一个标记为 inl 的PhysicalAddr, case 字段表达式将取第一个分支,绑定变量 x 为PhysicalAddr;第一个分支从 x 中抽取firstlast字段并返回它。类似地,如果 a 是一个标记为inr的值VirtualAddr,第二分支将被选择,并且送回VirtualAddr的name字段。这样,整个函数getName的类型是Addr→String。

上述的直观讨论形式地表述在图 11.9 中。对于项的语法,我们加上左投入、右投入以及构造 case;对于类型,可加上和构造子。对于求值,加入两个关于构造 case 的“beta 归约”规则——一个用于第一个子项已归约到一个标记为 inl 的值 v_0 的情况,另一个用于第一个子项已

① 像大部分变化类型的实际使用一样,这些例子也含有递归类型——一个列表的尾也是一个列表等。我们将在第 20 章中讨论递归类型。

② fullsimple 的实现实际上不支持我们这里描述的二元和的构造子——只支持下面将描述的更一般的变化。

归约到一个标记为 inr 的值 v_0 的情况；在每种情况中，选择适当的体，并且用 v_0 代换囿变量。其他求值规则在 case 的第一个子项中，并在 inl 和 inr 标记下执行求值。

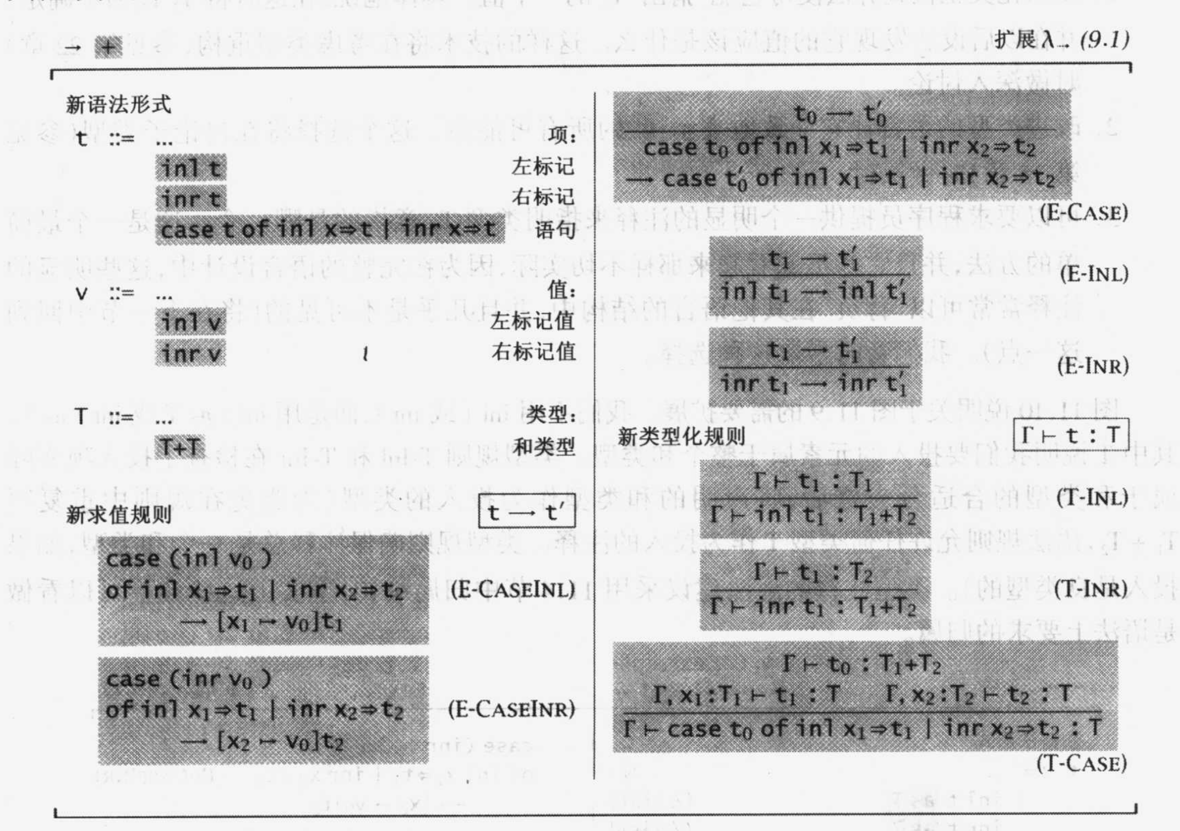


图 11.9 和

标记的类型规则是直接的：为了说明 inl t_1 有一个和类型 $T_1 + T_2$，只要说明 t_1 属于左边被加数 T_1，并且 inr 类似。对于 case 结构，首先检查第一个子项有一个和类型 $T_1 + T_2$，然后检查两个分支的体 t_1 和 t_2 有相同结果类型 T，假设它们囿变量 x_1 和 x_2 分别有类型 T_1 和 T_2；整个 case 的结果则是 T。遵循以前定义的约定，图 11.9 没有明确说明变量 x_1 和 x_2 的辖域是分支体 t_1 和 t_2，但这个事实可以从类型规则 T-Case 中扩展的上下文中得出。

11.9.1 练习[★★]：注意 case 的类型规则和图 8.1 中 if 的类型规则具有相似性：if 能看做是case的一个退化形式，其中没有信息传到分支上。请将true，false和 if 定义为和类型以及 Unit 类型的导出形式，形式化上面的说法。

类型的和以及惟一性

纯 $\lambda_{\rightarrow}$（参见 9.3 节）的类型关系的大部分性质可扩展到含和类型的系统中，但一个重要的性质则不能：即类型惟一性定理(9.3.3)。困难来自于标记结构 inl 和 inr。比如说，类型规则 T-Inl 表明：一旦我们已经说明 t_1 是 T_1 的一个元素，就能导出对任何类型T_2，inl t_1 是$T_1 + T_2$的一个元素。比如，我们能导出inl 5：Nat + Nat和inl 5：Nat + Bool（以及无限多个其他类型）。类型惟一性的不成立意味着我们不能简单地“自底向上读所有的规则”来建立一个类型检查算法，就

像我们一直这样做的那样。这时，可有几种选择：

1. 复杂化类型检查算法使得它能“猜出”T_2 的一个值。具体地说，在这时将 T_2 设为不确定，并在以后设法发现它的值应该是什么。这样的技术将在考虑类型重构（参见第 22 章）时做深入讨论。
2. 改进类型的语言让它一致地表示 T_2 的所有可能值。这个选择将在讨论子类型（参见第 15 章）时做探讨。
3. 可以要求程序员提供一个明显的注释来指明类型 T_2 意指的是哪一个。这是一个最简单的方法，并且它并不像看起来那样不切实际，因为在完整的语言设计中，这些明显的注释常常可以“背负”在其他语言的结构中，并且几乎是不可见的（将在下一节中回到这一点）。我们暂时采取这种选择。

图 11.10 说明关于图 11.9 的需要扩展。我们不用 inl t 或 inr t，而是用 inl t as T 或 inr t as T，其中 T 说明我们要投入的元素属于整个和类型。类型规则 T-Inl 和 T-Inr 在检查了投入项实际属于和类型的合适分支之后，将声明的和类型作为投入的类型（为避免在规则中重复写 $T_1 + T_2$，语法规则允许任何类型 T 作为投入的注释。类型规则确保注释总是一个和类型，如果投入是良类型的）。类型注释的语法建议采用 11.4 节中归属结构：事实上这些注释可以看做是语法上要求的归属。

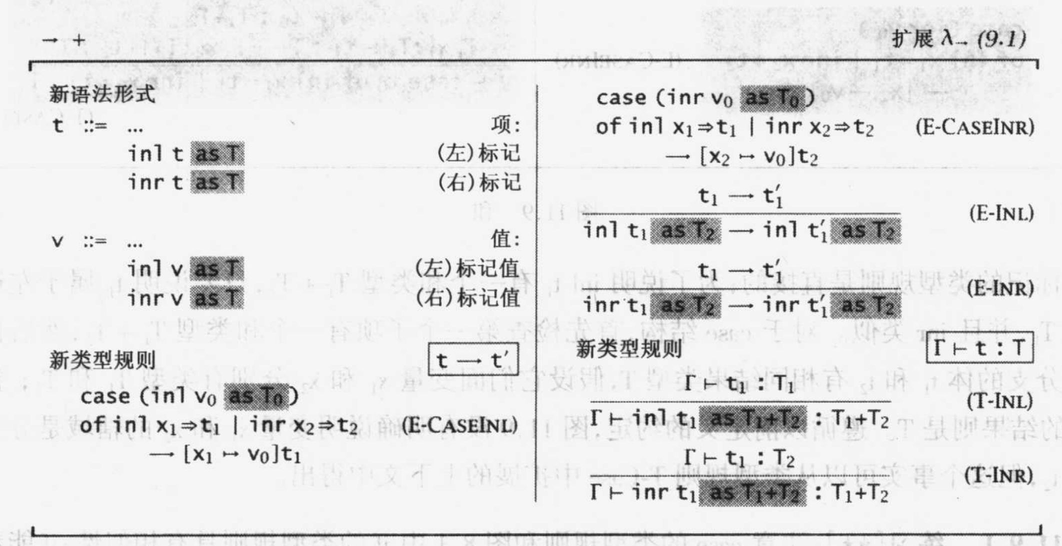


图 11.10 （带惟一类型的）和

11.10 变式

二元和推广为带标签的变式就像乘积推广为含标签的记录。我们不把它写成 $T_1 + T_2$，而是记为 $<l_1:T_1, l_2:T_2>$，其中 l_1 和 l_2 是字段标签。不用 inl t as $T_1 + T_2$，而用 $<l_1 = t>$ as $<l_1:T_1, l_2:T_2>$。不用给含 inl 和 inr 的 case 分支加上标签，而是用同一标签作为相应的和类型。有了这些推广，上节中的 getAddr 例子变成：

```
Addr = <physical:PhysicalAddr, virtual:VirtualAddr>;
a = <physical=pa> as Addr;
▸ a : Addr

getName = λa:Addr.
  case a of
    <physical=x> ⇒ x.firstlast
  | <virtual=y> ⇒ y.name;
▸ getName : Addr → String
```

变式的形式定义在图 11.11 中给出。注意，如同 11.8 中的记录一样，在一个变式的类型中标签的次序在这里是重要的。

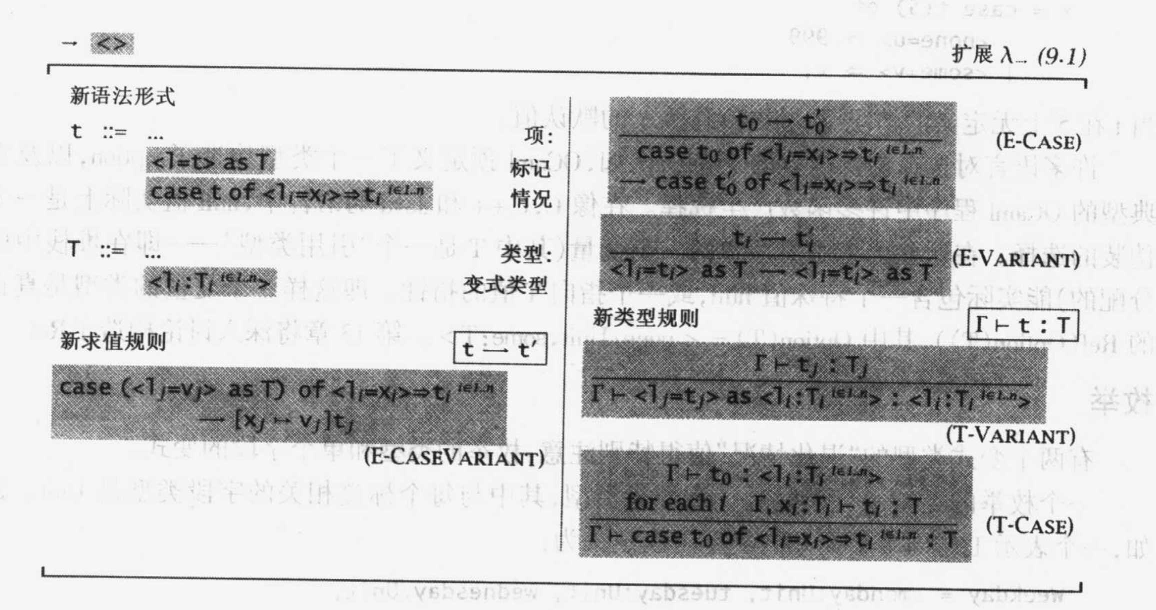


图 11.11 变式

选择

关于变式的一个有用的习惯用法是选值。比如，类型：

```
OptionalNat = <none:Unit, some:Nat>;
```

的一个元素要么是带有标记 none 的平凡 unit 值，要么是带有标记 some 的一个数。换言之，类型 OptionalNat 是同构于由一个附加不同值 none 扩充的 Nat。比如，类型：

```
Table = Nat→OptionalNat;
```

表示从数到数的有限映射：这样的一个映射的定义域是输入的集合，对某个 n 结果为 < some = n >。如果空表：

```
emptyTable = λn:Nat. <none=unit> as OptionalNat;
▸ emptyTable : Table
```

是一个常量函数，对每个输入返回 none。构造子：

```
extendTable =
  λt:Table. λm:Nat. λv:Nat.
    λn:Nat.
      if equal n m then <some=v> as OptionalNat
      else t n;
▸ extendTable : Table → Nat → Nat → Table
```

取一个表并加上(或覆盖)一个条件输入映射,将输入 m 映射到输出 <some = v> [equal 函数定义在练习(11.11.1)的答案中]。

可以通过封装 case 语句来对 Table 查找,从而得到结果。比如,如果 t 为所定义的表并且要查找条件 5 的情况,可以写为:

```
x = case t(5) of
      <none=u> ⇒ 999
    | <some=v> ⇒ v;
```

当 t 在 5 上无定义的情况下,将 999 作为 x 的默认值。

许多语言对选择提供内置的支持。比如,OCaml 预定义了一个类型构造子 option,以及在典型的 OCaml 程序中许多函数产生选择。在像 C,C++ 和 Java 的语言中,null 值实际上是一个伪装的选择。在这些语言中类型 T 的一个变量(其中 T 是一个"引用类型"——即在堆栈中所分配的)能实际包含一个特殊值 null,或一个指向 T 值的指针。即这样一个变量的类型是真正的 Ref(Option(T)),其中 Option(T) = <none:Unit,some:T>。第 13 章将深入讨论构造子Ref。

枚举

有两个变式类型的"退化情况"值得特别注意:枚举的类型和单个字段的变式。

一个枚举的类型(或枚举)是一个变式类型,其中与每个标签相关的字段类型是 Unit。比如,一个表示工作周中星期几的类型可以定义为:

```
Weekday = <monday:Unit, tuesday:Unit, wednesday:Unit,
           thursday:Unit, friday:Unit>;
```

这个类型的元素是 <monday = unit> as Weekday 这样的项。的确,因为类型 Unit 只有 unit 作为其成员,类型 Weekday 共有 5 个值,每个值对应一周的一天。构造子 case 可以用来定义枚举类型上的计算:

```
nextBusinessDay = λw:Weekday.
  case w of <monday=x>    ⇒ <tuesday=unit> as Weekday
          | <tuesday=x>   ⇒ <wednesday=unit> as Weekday
          | <wednesday=x> ⇒ <thursday=unit> as Weekday
          | <thursday=x>  ⇒ <friday=unit> as Weekday
          | <friday=x>    ⇒ <monday=unit> as Weekday;
```

显然,我们这里用的具体语法使得程序不易写和不易读。某些语言(最早是 Pascal)提供特殊语法来声明和使用枚举类型。其他语言(如 ML)将枚举作为变式的一个特殊情况。

单字段变式

另一个有意思的特殊情况是只带单个标签 l 的变式类型:

```
V = <l:T>;
```

这样的类型初看起来可能不是非常有用：毕竟 V 的元素将一一对应于字段类型 T 的元素，因为 V 的每个成员有形式 <l = t>（对某个 t:T）。重要的是在 T 上的通常操作在没有事先解包它们时，不可以应用到 V 的元素：一个 V 不可能碰巧为一个 T。

比如，假设我们写一个程序做多币种财务计算。这样的一个程序可以包括在美元和欧元之间的转换函数。如果两个表示为 Float，则这些函数看起来像这样：

```
dollars2euros = λd:Float. timesfloat d 1.1325;
▸ dollars2euros : Float → Float
euros2dollars = λe:Float. timesfloat e 0.883;
▸ euros2dollars : Float → Float
```

（其中 timesfloat: Float→Float→Float 乘以浮点数）如果我们从一个美元数目开始：

```
mybankbalance = 39.50;
```

可以转换为欧元，然后像这样：

```
euros2dollars (dollars2euros mybankbalance);
▸ 39.49990125 : Float
```

转换回美元。所有这些都有意义。但我们很容易执行完全没有意义的操作运算。比如，可以转换我的银行存款到欧元两次：

```
dollars2euros (dollars2euros mybankbalance);
▸ 50.660971875 : Float
```

因为所有的数目简单地表示为浮点，没有办法可以帮助类型系统阻止这种无意义的行为。尽管如此，如果我们定义美元和欧元为不同的变式类型(它们的根本表示是浮点型)：

```
DollarAmount = <dollars:Float>;
EuroAmount = <euros:Float>;
```

则可以定义转换函数的安全形式，它们将只接受当前币种的数目。

```
dollars2euros =
  λd:DollarAmount.
    case d of <dollars=x> ⇒
      <euros = timesfloat x 1.1325> as EuroAmount;
▸ dollars2euros : DollarAmount → EuroAmount
euros2dollars =
  λe:EuroAmount.
    case e of <euros=x> ⇒
      <dollars = timesfloat x 0.883> as DollarAmount;
▸ euros2dollars : EuroAmount → DollarAmount
```

现在类型检查器可以跟踪在计算中用到的币种，并且提醒我们如何解释最终结果：

```
mybankbalance = <dollars=39.50> as DollarAmount;
euros2dollars (dollars2euros mybankbalance);
▸ <dollars=39.49990125> as DollarAmount : DollarAmount
```

此外，如果写一个无意义的两次转换，类型将不能匹配，并且我们的程序(即使正确)将被拒绝：

```
dollars2euros (dollars2euros mybankbalance);
▸ Error: parameter type mismatch
```

变式和数据类型

一个形为 $<l_i:T_i{}^{i\in 1..n}>$ 的变式类型 T 大致类似于 ML 中的形为，定义为①：

```
type T = l₁ of T₁
       | l₂ of T₂
       | ...
       | lₙ of Tₙ
```

的数据类型。但它们之间存在几个差别值得注意。

1. 不重要但可能引起混淆的是，我们这里假设的标识符的大小写约定不同于 OCaml 的相应约定。在 OCaml 中类型必须以小写字母开头，并且数据类型构造子（用我们的术语，标签）以大写字母开头，因此，严格地讲，上述的数据类型声明应该写成如下形式：

   ```
   type t = L₁ of t₁ | ... | Lₙ of tₙ
   ```

   为避免项 t 和类型 T 之间的混淆，在下面的讨论中将忽略 OCaml 的约定，而采用我们的约定。

2. 最有趣的差别是当一个构造子 l_i 用于将 T_i 的一个元素投入到一个数据类型 T 时，OCaml 不要求类型注释：简单地写 l_i(t)。OCaml 处理这个的办法（并保留惟一类型）是数据类型 T 必须在它能使用之前声明的。此外，在 T 中的标签不能被任何其他在同一辖域中声明的数据类型所使用。因此，当类型检查器看见 l_i(t)时，它知道注释只能是 T。事实上，注释是隐藏在标签中的。

   这个技巧消去了许多愚蠢的注释，但它导致了用户之间一定的抱怨，因为它意味着标签不能在不同数据类型之间共享——至少不能在同一模块内共享。在第 15 章中，我们将看到另一种方法来省略注释以避免这种缺点。

3. 另一个 OCaml 采用的方便技巧是一个数据类型定义中相关的类型就是 Unit 时，它可以一起忽略掉。例如可定义为：

   ```
   type Weekday = monday | tuesday | wednesday | thursday | friday
   ```

   而不用：

   ```
   type Weekday = monday of Unit
                | tuesday of Unit
                | wednesday of Unit
                | thursday of Unit
                | friday of Unit
   ```

   类似地，标签 monday 本身（而不是应用于平凡值 unit 的 monday）不认为是类型 Weekday 的一个值。

---

① 为了与本书中其他实现一致起见，这一节采用 OCaml 具体的数据类型语法，但它们出现在 ML 的较早版本中，可以在 Standard ML 以及 Haskell 这样类似 ML 语言中找到基本上相同形式。数据类型和模式匹配是这些日常程序设计语言的一个最有用的优点。

4. 最后,OCaml 数据类型实际上将变式类型与几个我们将在以后章节中分别讨论的附加特征绑在一起。

- 一个数据类型定义可以是递归的——即定义的类型允许出现在定义体中。比如,在 Nat 列表的标准定义中,标记为 cons 的值是一个序对,它的第二个元素是一个 NatList:

```
type NatList = nil
             | cons of Nat * NatList
```

- 一个 OCaml 数据类型可以在一个类型变量上参数化,如同在数据类型 List 的一般定义中一样:

```
type 'a List = nil
             | cons of 'a * 'a List
```

从类型理论的角度来看,List 可以看做是一种函数(称为一个类型算子),它将'a 的每个选择映射到一个具体的数据类型,Nat 到 NatList 等。类型算子是第 29 章的主题。

作为不交并的复式

和类型与变式类型有时称为不交并。类型 $T_1 + T_2$ 是 T_1 和 T_2 的一个“并”,意义为:它的元素包括 T_1 和 T_2 的所有元素。这个并是不相交的,因为 T_1 或 T_2 的元素集合是在它们组合之前分别标记为 inl 或 inr,使得并的元素很清楚,是来自 T_1 还是来自 T_2。词语并类型也用来指无标记(相交)并类型,将在 15.7 节中描述。

动态类型

即使在静态类型语言中,常常要求处理那些类型在编译时不能确定的数据,特别是出现在数据的存活时间跨越多个机器或编译器的多个执行,比如当数据存储在一个外部文件系统或数据库中,或通过一个网络通信时。为了安全地处理这样的情形,许多语言提供工具在执行时间来检查值的类型。

实现这一点的一个办法是加上一个类型Dynamic,它的值是一个值 v 和一个类型 T 的序对,其中 v 有类型 T。Dynamic的实例用一个明显的标记结构来建立,并且用一个类型安全 typecase 结构来检查。事实上,Dynamic可以看做是一个无限不交并,它的标签是类型。参见 Gordon(circa 1980),Mycroft (1983),Abadi,Cardelli,Pierce和Plotkin (1991b),Leroy和Mauny(1991),Abadi,Cardelli,Pierce和Rémy(1995),以及Henglein(1994)。

11.11 一般递归

在大部分程序语言中可以找到的另一个功能是定义递归函数的能力。我们已经见到(在第 5 章中)在无类型 lambda 演算中,这样的函数可以借助于 fix 组合式来定义。

递归函数可以用类似的方法在一个类型环境中定义。比如,这里 iseven 函数,当用一个偶数参数调用时它返回 true,否则返回 false:

```
ff = λie:Nat→Bool.
       λx:Nat.
         if iszero x then true
         else if iszero (pred x) then false
         else ie (pred (pred x));
```

```
▸ ff : (Nat→Bool) → Nat → Bool

  iseven = fix ff;

▸ iseven : Nat → Bool

  iseven 7;

▸ false : Bool
```

直观地,传给 fix 的高阶函数 ff 是函数 iseven 的一个产生器:如果 ff 运用于一个函数 ie,这个函数模拟 iseven 的行为最大到数 n(即对小于或等于 n 的输入返回正确结果的一个函数),然后它返回一个更接近 iseven 的函数,对一直到 $n+2$ 的输入都能返回正确结果。运用 fix 到这个产生器返回它的不动点函数,对所有的输入 n 做出相应的行为。

然而,有一个重要的差别不同于无类型环境:fix 本身不能定义在简单类型 lambda 的演算中。的确,在第 12 章中,将看到没有能导致非终止性计算的表达式,可以只用简单类型来类型化[①]。因此,我们不是定义 fix 为语言中的一个项,而是简单地将它加上,作为一个新原语,其求值规则模仿无类型 fix 组合式的行为,以及用一个类型规则捕捉它想要的用法。这些规则在图 11.12 中列出(后面会谈到 letrec 缩写)。

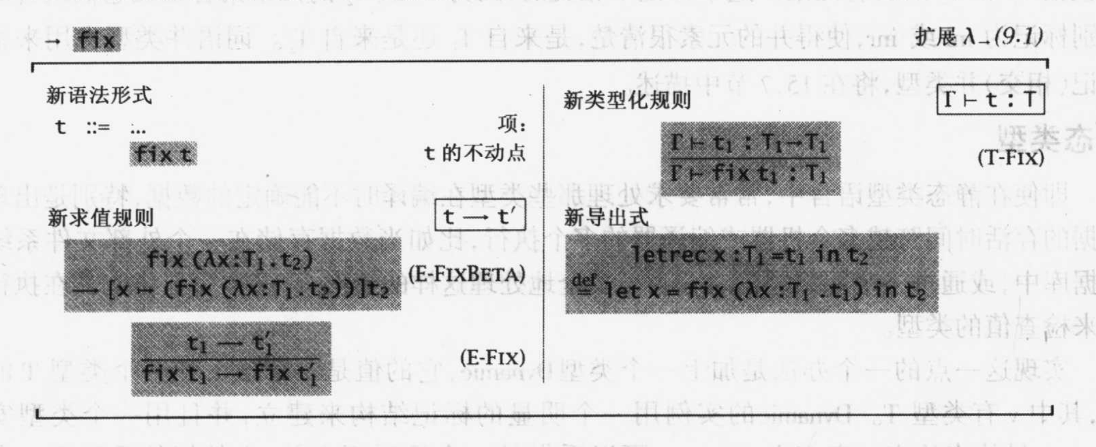


图 11.12 一般递归

具有数和 fix 的简单类型 lambda 演算,长期以来是程序语言研究者所喜欢的实验主题,因为它是一些复杂的语义现象中最简单的语言。这类语义现象中包括了全抽象(Plotkin,1977,Hyland 和Ong,2000,Abramsky,Jagadeesan和Malacaria,2000)。它常常被称为 *PCF*。

11.11.1 练习[★★]:用 fix 定义equal,plus,times 和 factorial。

fix 结构典型地用来建立函数(如函数到函数的函数不动点),但值得注意的是在规则 T-Fix 中的类型 T 不限于函数类型。这种额外的能力有时是有用的。比如,它允许我们定义互递归函数的一个记录为在(函数的)记录上的一个函数的不动点。如下面的iseven的实现用到一个辅助函数isodd;两个函数定义为一个记录的字段,其中这个记录的定义在一个记录 ieio 上抽象,它的成员用来从 iseven 和 isodd 字段体中做递归调用。

---

① 在以后章节中(参见第 13 章和第 20 章),将看到一些恢复系统中 fix 定义的简单类型的扩展。

```
ff = λieio:{iseven:Nat→Bool, isodd:Nat→Bool}.
        {iseven = λx:Nat.
                    if iszero x then true
                    else ieio.isodd (pred x),
         isodd = λx:Nat.
                    if iszero x then false
                    else ieio.iseven (pred x)};

▸ ff : {iseven:Nat→Bool,isodd:Nat→Bool} →
       {iseven:Nat→Bool, isodd:Nat→Bool}
```

形成函数 ff 的不动点给出两个函数的一个记录：

```
r = fix ff;

▸ r : {iseven:Nat→Bool, isodd:Nat→Bool}
```

并且这些函数的第一投影是函数 iseven 本身：

```
iseven = r.iseven;

▸ iseven : Nat → Bool

iseven 7;

▸ false : Bool
```

能对任何 T 的一个类型为 T→T 的函数形成其不动点，会带来某种惊人的结果。特别是，它表明了每个类型都拥有一些项。为了明白这一点，注意，对每个类型 T，我们能定义一个函数 $diverge_T$ 如下所示：

```
divergeT = λ_:Unit. fix (λx:T.x);

▸ divergeT : Unit → T
```

当 $diverge_T$ 运用于一个 unit 参数，可得到一个非终止性求值序列，对此序列一次一次地运用 E-FixBeta，总会产生相同的项。即对每个类型 T，项 $diverge_T$ unit 是 T 的一个无定义元素。我们可以考虑的（最后的）改进是绑定一变量为一个递归定义的结果，为这样的普通情况引入更方便具体的语法。在大部分高级语言中 iseven 的第一个定义可以写成如下形式：

```
letrec iseven : Nat→Bool =
  λx:Nat.
    if iszero x then true
    else if iszero (pred x) then false
    else iseven (pred (pred x))
in
  iseven 7;

▸ false : Bool
```

递归绑定构造子 letrec 可以容易地定义为一个导出形式：

$$\texttt{letrec } x{:}T_1{=}t_1 \texttt{ in } t_2 \quad \stackrel{\text{def}}{=} \quad \texttt{let } x = \texttt{fix } (\lambda x{:}T_1.t_1) \texttt{ in } t_2$$

11.11.2 **练习**[★]：用letrec而不用fix，重写练习（11.11.1）中plus，times和factorial的定义。

Klop（1980）和Winskel（1993）进一步讨论了不动点算子。

11.12 列表

我们见到的类型特征可以分为基本类型，如Bool，Unit 及类型构造子，如→和 ×，用来从旧

的类型建立新的类型。另一个有用的类型构造子是List。对每个类型 T,类型List T描述有限长且元素来自 T 的列表。

图 11.13 给出了列表的语法、语义和类型规则。除了语法差别(是 List T 而不是 T list 等)以及对我们表示中的所有语法形式的显式类型注释外,这些列表基本上等同于在 ML 和其他函数式语言中的列表(类型 T 的元素)。空列表记为 nil[T]。加一个(类型 T 的)新元素 t_1 到一个列表 t_2 的前部所形成的列表记为 cons[T]$t_1 t_2$。一个列表 t 的头和尾记为 head[T]t 和 tail[T]t;布尔谓词 isnil[T]t 产生 true 当且仅当 t 是空的[①]。

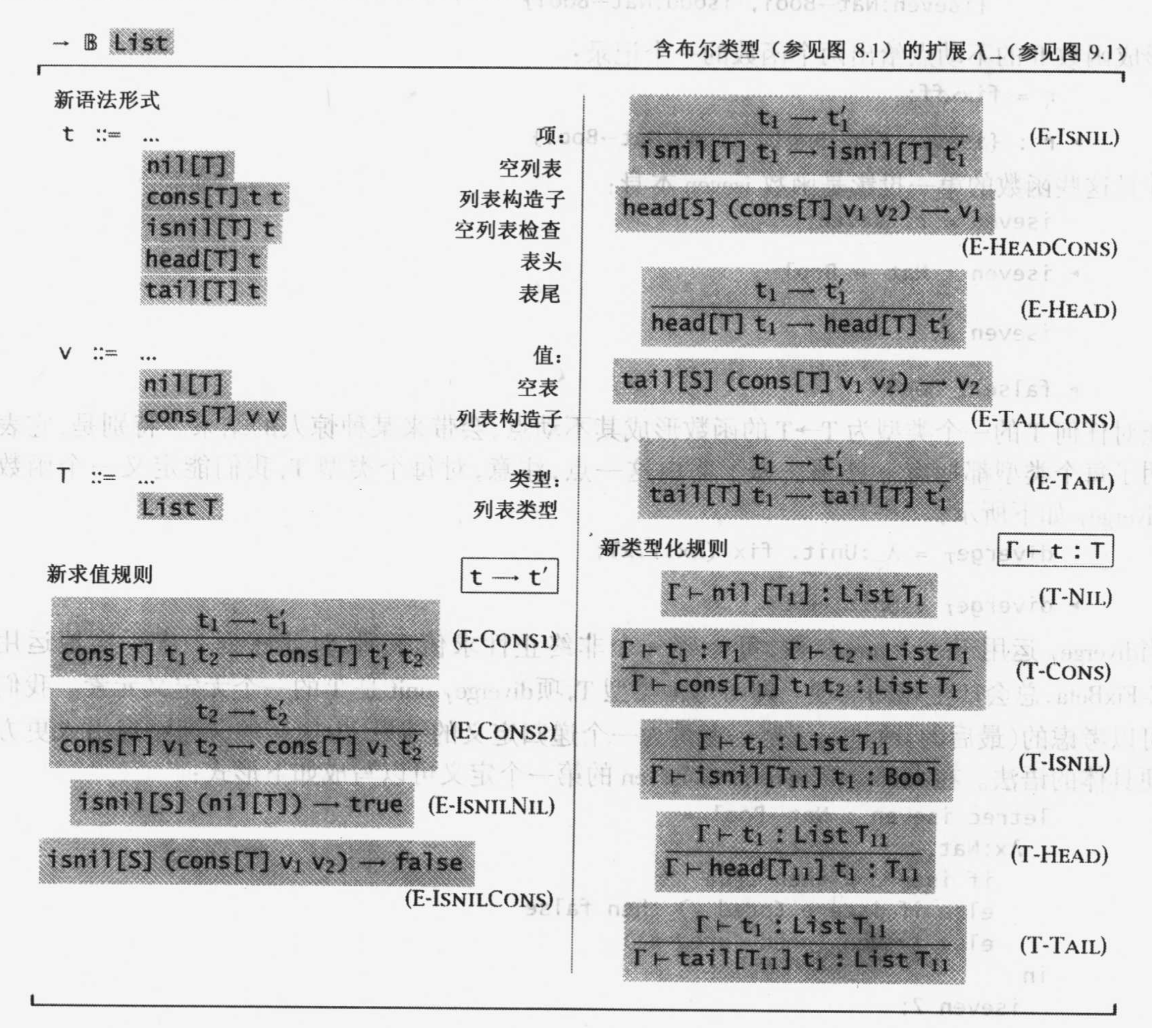


图 11.13 列表

11.12.1 练习[★★★]:验证带布尔型和列表型的简单类型 lambda 演算的进展和保持定理。

11.12.2 练习[★★]:这里列表的表示包括许多实际不需要的类型注释,因为类型规则可以很容易地由上下文导出注释(这些注释打算用来简化与在 23.4 节中列表编码之间的比较)。所有的类型注释可以删除吗?

---

① 为了简单起见,我们这里采用列表的"head/tail/isnil 表示"。从语言设计角度来看,将列表作为一个数据类型,用 case 表达式来分解可能更好,因为更多的程序错误可以这样作为类型错误捕捉到。

第12章 规 范 化[①]

在本章中，我们将考虑纯简单类型 lambda 演算的另一个基本理论性质：一个良类型程序的求值保证在有限步内停机(即每个良类型项是可规范化的)。

不像我们已经考虑过的类型安全性性质，规范化性质不能扩展到完整的程序语言，因为这些语言几乎总是扩展简单类型 lambda 演算，使其带有像一般递归(参见 11.11 节)或递归类型(参见第 20 章)这样的构造子，有了它们就可以写非终止性程序。尽管如此，规范化的问题在讨论(参见 30.3 节)系统 F_ω 的元理论时，在类型的层次上将会重新提起，在这个系统中，类型的语言有效地包含简单类型 lambda 演算的一个拷贝，并且类型检查算法的终止与否，将由类型表达式上的一个“规范化”操作停止来决定。

研究规范化证明的另一个理由是，它们是在类型论中发现的最优美(令人兴奋的)的数学，常常(这里所见)包含逻辑关系的基本证明技巧。

某些读者在第一次读本书时可以跳过这一章；这样做不会影响阅读以后的章节 。

12.1 简单类型的规范化

这里将考虑的演算是在一单个基本类型 A 上的简单类型 lambda 演算。这个演算的规范化不是完全平凡的，因为一个项的每个归约可以在子项中复制约式。

12.1.1 练习[★]：如果我们企图直接对一个良类型项的长度做规范化归纳证明，会在什么地方失败呢？

这里的关键问题(如在许多归纳证明中一样)是找一个足够强的归纳假设。为此，我们对每个类型 T，定义一个类型 T 的封闭项集合 R_T。我们把这些集合看做是谓词，并且用 $R_T(t)$ 表示 $t \in R_T$[②]。

12.1.2 定义：

- $R_A(t)$ 当且仅当 t 停止。
- $R_{T_1 \to T_2}(t)$ 当且仅当 t 停止，并且当 $R_{T_1}(s)$ 时，我们有 $R_{T_2}(t\ s)$。

这个定义给出了需要的归纳假设。我们的主要目标是证明所有的程序(即基本类型的所有封闭项)停止。但基本类型的封闭项可以包含函数类型的子项，这样就需要知道关于子项的基本性质。此外，光知道这些子项停止还是不够的，因为一个规范化的函数对一个规范化参数的运用包含一个代换，这可能需要更多步的求值。因此我们对函数类型的项需要一个更强的条件：它们不仅本身停止，而且运用于停止的参数时，也将产生停止的结果。

---

① 本章中讨论的语言是带单个基本类型 A(参见图 11.1)的简单类型 lambda 演算(如图 9.1 所示)。

② 集合 R_T 有时称为饱和集合或可归约候选。

定义(12.1.2)的形式是逻辑关系证明技巧的特征(因为,这里只处理一元关系,应该称逻辑谓词更合适)。如果我们要证明所有类型A的封闭项具有某个性质 P,对类型做归纳,证明所有类型A的项具有性质 P,所有类型 $A \to A$ 的项保持性质 P,所有类型 $(A \to A) \to (A \to A)$ 的项都有保持 P 性质的性质,等等。我们可以通过定义由类型为索引的谓词集合来做到这一点。对于基本类型A,谓词就是 P。对于函数类型,函数应该将满足输入类型谓词的值映射到满足输出类型谓词的值。

我们使用这个定义在两步内给出规范化的证明。首先,注意到每个集合 R_T 的每个元素是可规范化的。然后证明类型T的每个良类型的项是 R_T 的一个元素。

第一步直接由 R_T 的定义得到。

12.1.3 引理:如果 $R_T(t)$,则t停止。

第二步分为两个引理。首先,注意 R_T 中的成员关系在求值下保持不变。

12.1.4 引理:如果 $t:T$ 并且 $t \to t'$,则 $R_T(t)$,当且仅当 $R_T(t')$。

证明:对类型T的结构做归纳。首先,这是很清楚的:t停止,当且仅当t′停止。如果 $T=A$,没有什么需要证明的。另一方面,假设对某个 T_1 和 T_2,有 $T = T_1 \to T_2$。对于"仅当"部分(⇒),假定 $R_T(t)$ 以及对某个任意的 $s:T_1$,$R_{T_1}(s)$。由定义,可有 $R_{T_2}(t\ s)$。但 $t\ s \to t'\ s$,由对类型 T_2 的归纳假定,我们有 $R_{T_2}(t'\ s)$。因为这对任意一个s成立,R_T 的定义给出 $R_T(t')$。对于"当且"部分(⇐)的讨论是类似的。

下面,我们要证明类型T的每个项属于 R_T。这里,将对类型推导做归纳(如果看到一个关于良类型项的证明不是对类型推导做归纳可能会令人感到惊讶)。在这里,困难之处在于处理λ抽象的情况。因为我们是用归纳方法证明,证明一个项 $\lambda x{:}T_1.t_2$ 属于 $R_{T_1 \to T_2}$ 将包括运用归纳假设来证明 t_2 属于 R_{T_2}。但是 R_{T_2} 定义为封闭项的集合,而 t_2 可以包含某个自由的x,这样就没有意义了。

这个问题可以用适当推广的归纳假设这个技巧来解决:我们不是证明含有一个封闭项的语句,而是推广它使得包含一个开项t的所有封闭实例。

12.1.5 引理:如果 $x_1{:}T_1,\cdots,x_n{:}T_n \vdash t:T$ 并且 $v_1,\cdots,v_n$ 分别是类型 $T_1,\cdots,T_n$ 的封闭值,使得对每个 i,有 $R_{T_i}(v_i)$,则 $R_T([x_1 \mapsto v_1]\cdots[x_n \mapsto v_n]t)$。

证明:对 $x_1{:}T_1,\cdots,x_n{:}T_n \vdash t:T$ 的一个推导做归纳(最有意思的情况是抽象)。

情况 T-VAR: $t = x_i \qquad T = T_i$

证明是直接的。

情况 T-ABS: $t = \lambda x{:}S_1.s_2 \qquad x_1{:}T_1,\ \ldots,\ x_n{:}T_n,\ x{:}S_1 \vdash s_2 : S_2$
$T = S_1 \to S_2$

显然,$[x_1 \mapsto v_1]\cdots[x_n \mapsto v_n]t$ 求值为一个值,因为它早已是一个值。下面要证明的是对任何 $s{:}S_1$ 使得 $R_{S_1}(s)$,有 $R_{S_2}(([x_1 \mapsto v_1]\cdots[x_n \mapsto v_n]t)s)$。假设s是这样一个项。由引理(12.1.3),我们对某个v有 $s \to^* v$。由引理(12.1.4),$R_{S_1}(v)$。现在,由归纳假设,$R_{S_2}([x_1 \mapsto v_1]\cdots[x_n \mapsto v_n][x \mapsto v_1]S_2)$。但:

$$(\lambda x{:}S_1.\ [x_1 \mapsto v_1]\cdots[x_n \mapsto v_n]s_2)\ s$$
$$\longrightarrow^*\ [x_1 \mapsto v_1]\cdots[x_n \mapsto v_n][x \mapsto v]s_2,$$

引理(12.1.4)告诉我们:

$$R_{S_2}((\lambda x{:}S_1.\ [x_1 \mapsto v_1]\cdots[x_n \mapsto v_n]s_2)\ s),$$

即:$R_{S_2}((([x_1 \mapsto v_1]\cdots[x_n \mapsto v_n](\lambda x{:}S_1.\ s_2))s)$。因为 s 是任意选择的,由 $R_{S_1\to S_2}$ 的定义,得到:

$$R_{S_1\to S_2}([x_1 \mapsto v_1]\cdots[x_n \mapsto v_n](\lambda x{:}S_1.\ s_2)).$$

情况 T-APP: $t = t_1\ t_2$
$x_1{:}T_1, \cdots, x_n{:}T_n \vdash t_1 : T_{11}\to T_{12}$
$x_1{:}T_1, \cdots, x_n{:}T_n \vdash t_2 : T_{11}$
$T = T_{12}$

由归纳假设得到,$R_{T_{11}\to T_{12}}([x_1 \mapsto v_1]\cdots[x_n \mapsto v_n]t_1)$ 并且 $R_{T_{11}}([x_1 \mapsto v_1]\cdots[x_n \mapsto v_n]t_2)$。由 $R_{T_{11}\to T_{12}}$ 的定义:

$$R_{T_{12}}(([x_1 \mapsto v_1]\cdots[x_n \mapsto v_n]t_1)\ ([x_1 \mapsto v_1]\cdots[x_n \mapsto v_n]t_2)),$$

即 $R_{T_{12}}([x_1 \mapsto v_1]\cdots[x_n \mapsto v_n](t_1\ t_2))$,.

我们在引理(12.1.5)中简单地取项 t 为封闭的,然后记住对每个 T,R_T 的所有元素是规范的,作为一个推论我们就得到了规范化的性质。

12.1.6 定理[规范化]:如果$\vdash$ t:T,则 t 是可规范化的。

证明:由引理(12.1.5),有 $R_T(t)$;因此,由引理(12.1.3)中,t 是可规范化的。

12.1.7 练习[推荐,★★★]:扩展本章中的证明技巧来证明简单类型 lambda 演算在扩展为包含布尔值(参见图 3.1)和乘积(参见图 11.5)后保持可规范性。

12.2 注释

在许多文献中将规范化性质形式化为带完全(非确定性的)beta 归约的演算的强规范化。Tait(1967)提出了标准证明方法,Girard(1972,1989)将它推广到系统 F(参见第 23 章),以后Tait(1975)简化了证明。这里采用的形式是将 Tait 的方法改为按值调用的方法,该方法是由Martin Hofmann提出的(私人通信)。逻辑关系证明技巧的经典文献参见Howard (1973),Tait(1967),Friedman(1975),Plotkin (1973,1980),以及Statman (1982,1985a,1985b)。在许多关于语义的文献中,如Mitchell(1996)和Gunter(1992),也有讨论。

Tait 的强规范化证明对应于所谓的求值规范化或类型导向部分求值的简单类型项求值的算法(Berger,1993; Danvy,1998);也可参见Berger和Schwichtenberg(1991),Filinski(1999),Filinski(2001)以及Reynolds(1998a)。

第13章 引 用[1]

到目前为止,我们讨论了各种纯语言特征,包括函数抽象,如数和布尔值的基本类型,记录和变式的结构化类型。这些特征形成了大部分程序语言的主干——包括像 Haskell 这样的纯函数式语言,ML 这样的“主要函数式”语言, C 这样的命令式语言,以及 Java 这样的面向对象语言。

大部分实际程序语言也包含各种不纯的特征,它们不能在我们所用的简单语义框架中描述。特别地,除了只产生结果,在这些语言中项的求值可以指派给易变化的变量(引用单元、列表、易变化记录字段等),执行输入和输出到文件,显示或网络连接,通过异常,跳跃或继续控制非局部转换;参与处理内部的同步和通信等。在程序语言的文献中,计算的这种“副作用”通常称为计算效应。

在本章中,将看到一种计算效应(易变化引用)如何加到我们研究的演算中去。主要的扩展将明确地处理一个存储(或堆栈)。这个扩展是可以直接定义的;最有意思的部分是我们对类型保持定理[定理(13.5.3)]需要做的改进工作。考虑另一种效应——异常和控制的非局部转换(参见第 14 章)。

13.1 引言

几乎每个程序语言[2]提供某种形式的赋值操作,用来改变原先分配存储段的内容。在某些语言(如 ML 和它的类似语言)中,名称绑定机制和赋值机制是分开的。我们可以有一个变量 x,它的值是 5,或一个变量 y,它的值是一个引用(或指针),指向一个易变化的单元,这个单元的当前内容是 5,并且这个差别对程序员是可见的。可以将 x 加到另一个数,但不能给它赋值。我们可以用 y 直接将一个值赋给 y 所指向的单元(记 y: = 84),但不能用它直接作为 plus 的一个参数。必须明显将反引用记为“!y”,以得到它的当前内容。在大部分其他语言(特别在包括 Java 在内的 C 系列语言中)中,每个变量名称指向一个易变化的单元,并且反引用一个变量得到它当前内容的操作是隐式的[3]。

为了形式研究的目的,将这两个机制分开是有意义的[4];在本章中,将研究的模型接近于 ML 模型。这些内容可直接应用到像 C 这样的语言中,只要忽略某些差别,将一些操作,如把反引用看做隐式的,而不是显式的。

---

① 本章中讨论的系统是带 Unit 和引用的简单类型 lambda 演算(如图 13.1 所示)。相应的 OCaml 实现是 fullref。

② 即使像 Haskell 这样通过单体扩展的“纯函数式”语言。

③ 严格地讲,在 C 或 Java 中类型 T 的大多数变量实际上看做是指向包含类型为 Option(T)值的单元指针,说明一个变量的内容可以是一个合适的值或者是一个特殊值 null。

④ 从语言设计的角度来看这种分开处理也是值得的。让易变单元以明确的选择方式来使用,而不是采取默认的方式,可以鼓励采用主函数式程序设计风格,这种风格中引用很少使用;这种办法可以使程序更易写、维护和推理,特别在像并发性这样的特征出现的时候。

基础

引用的基本操作是分配、反引用和赋值。为了分配一个引用,我们用算子 ref,对新的单元提供一个初始值:

```
r = ref 5;
▸ r : Ref Nat
```

类型检查器会指出 r 的值是一个引用,指向一个总是含有一个数的单元。为了读这个单元中的当前值,我们用反引用算子"!":

```
!r;
▸ 5 : Nat
```

为了改变存储在这个单元中的值,使用赋值算子:

```
r := 7;
▸ unit : Unit
```

(赋值的结果是平凡的 unit 值;参见 11.2 节)如果再次对 r 反引用,可得到更新的值:

```
!r;
▸ 7 : Nat
```

副作用和序列

一个赋值表达式的结果是平凡值 unit 的事实与定义在 11.3 节中的序列概念非常符合:序列允许写:

```
(r:=succ(!r); !r);
▸ 8 : Nat
```

来替代等价的但冗长的形式:

```
(λ_:Unit. !r) (r := succ(!r));
▸ 9 : Nat
```

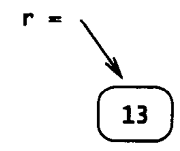
并且按次序求值两个表达式并送回第二个的值。限制第一个表达式的类型为 Unit 让第一个值保证平凡时将其丢弃,这样可帮助类型检查器捕捉一些简单的错误。

注意,如果第二个表达式也是一个赋值,则整个序列的类型将是 Unit,因此我们能有效地将它放在另一个表达式的左边来建立更长的赋值序列:

```
(r:=succ(!r); r:=succ(!r); r:=succ(!r); r:=succ(!r); !r);
▸ 13 : Nat
```

引用和别名

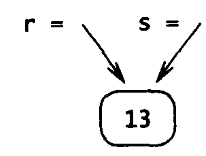
记住囿界于 r 的引用和由这个引用所指的存储中的单元之间的差别是很重要的:



如果我们做 r 的一个拷贝,使得绑定它的值为另一个变量 s:

```
s = r;
▸ s : Ref Nat
```

所拷贝的只是引用(图中的箭头),而不是单元:



我们可以验证这一点,通过赋一个新的值到 s:

```
s := 82;
▸ unit : Unit
```

并且通过 r 读它:

```
!r;
▸ 82 : Nat
```

引用 r 和 s 称为是同一个单元的别名。

13.1.1 练习[★]:画一个类似的图来说明求值表达式 a = {ref 0, ref 0} 和 b = (λx:Ref Nat. {x,x})(ref 0)的效果。

共享状态

可以使用别名使得带引用的程序推理非常困难。比如,表达式(r: = 1; r: = !s),赋 1 给 r,并且马上用 s 当前的值覆盖它,与直接赋值 r: = !s 有着相同效果,除非我们将它写在一个上下文,其中 r 和 s 是相同单元的别名。

当然,别名是使引用发挥作用的重要原因。特别是它允许我们在一个程序的不同部分之间设定"隐式通信通道"——共享状态。比如,假设我们定义一个引用单元和两个函数来处理它的内容:

```
c = ref 0;
▸ c : Ref Nat
incc = λx:Unit. (c := succ (!c); !c);
▸ incc : Unit → Nat
decc = λx:Unit. (c := pred (!c); !c);
▸ decc : Unit → Nat
```

调用 incc:

```
incc unit;
▸ 1 : Nat
```

使 c 改变的原因可以通过调用 decc 观察到:

```
decc unit;
▸ 0 : Nat
```

如果将 incc 和 decc 包装在一个记录中：

```
    o = {i = incc, d = decc};
  ▸ o : {i:Unit→Nat, d:Unit→Nat}
```

则可以将整个结构作为一个单位，并用它的成员对在 c 中状态的共享部分做增量和减量操作。其实我们已经构造了一个简单对象。这个想法将在第 18 章中做进一步的讨论和研究。

对复合类型的引用

一个引用单元不一定只包含一个数：上述的原语允许我们产生指向任何类型值（包括对函数）的引用。比如，我们可以用到函数的引用，来给出一个（不是非常有效的）数列的实现。用 NatArray 表示类型 Ref(Nat→Nat)：

```
  NatArray = Ref (Nat→Nat);
```

为了建立一个新的数组，分配一个引用单元，并填入一个函数。这样当给定一个索引，总是返回 0：

```
    newarray = λ_:Unit. ref (λn:Nat.0);
  ▸ newarray : Unit → NatArray
```

为了查找一个数组中的元素，只要运用函数到相应的索引：

```
    lookup = λa:NatArray. λn:Nat. (!a) n;
  ▸ lookup : NatArray → Nat → Nat
```

编码有意思的部分是更新函数 update。它取一个数组、一个索引和一个新的存储在该索引位置中的值，并产生（存储在引用中）一个新的函数，这个函数当问起在这个索引上的值时，返回赋给 update 的新值，并且对所有其他的索引，将查询传给引用中以前存储的函数：

```
    update = λa:NatArray. λm:Nat. λv:Nat.
               let oldf = !a in
               a := (λn:Nat. if equal m n then v else oldf n);
  ▸ update : NatArray → Nat → Nat → Unit
```

13.1.2　练习[★★]：如果我们更紧致地定义 update 如下：

```
  update = λa:NatArray. λm:Nat. λv:Nat.
             a := (λn:Nat. if equal m n then v else (!a) n);
```

它的行为是否一样？

对包含其他引用的值再引用也很有用，它允许我们定义像易可变列表和树这样的数据结构（这样的结构通常包含递归类型，将在第 20 章中引入递归类型）。

垃圾收集

在我们形式化引用之前的最后一个问题是存储的回收。当引用单元不再需要时，并没有提供任何回收它的原语。的确，像许多现代语言（包括 ML 和 Java），我们依赖于执行时间系统来执行垃圾收集，即收集和重新使用程序不再访问的单元。这不仅是程序设计风格的问题：回收操作很难保证类型的安全性。主要是由摇摆引用问题引起的：我们分配含有一个数的单元，将指向它的引用存储在某个数据结构中，使用一会儿，然后回收它，再分配一个含布尔值的新

单元，可能重新使用同一个存储单元。这样就对同一个存储单元取了两个名称，一个类型为 Ref Nat，另一个类型为 Ref Bool。

13.1.3 练习[★★]：说明这如何违反类型安全性。

13.2 类型化

ref，：= 和！的类型规则直接根据我们对它们的行为描述得出：

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \texttt{ref}\ t_1 : \texttt{Ref}\ T_1} \quad \text{(T-Ref)}$$

$$\frac{\Gamma \vdash t_1 : \texttt{Ref}\ T_1}{\Gamma \vdash !t_1 : T_1} \quad \text{(T-Deref)}$$

$$\frac{\Gamma \vdash t_1 : \texttt{Ref}\ T_1 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 := t_2 : \texttt{Unit}} \quad \text{(T-Assign)}$$

13.3 求值

当我们考虑如何形式化引用的操作行为时出现了一个更复杂的现象。要想知道为什么要问："类型 Ref T 的值将是什么？"需要考虑的关键是求值一个 ref 算子还需要做点别的什么事（比如，分配某个存储），并且该操作的结果将是对这个存储的引用。

那么，一个引用是什么？

在大部分程序语言实现中执行时间存储，本质上就是一个大的字节数组。执行时间系统找到这个数组中当前使用的部分；当需要分配一个新的引用单元时，我们从存储的自由区域中分配一个足够大的段（一个整数占 4 个字节，浮点数占 8 个字节等），标记它为正在使用，然后送回新近分配区域的开头索引（典型地，一个 32 位或 64 位的整数）。这些索引就是引用。

我们暂时不需要太具体。

考虑存储时，应将不同值运行时表示的不同大小抽象出来，不将其考虑为一组字节而是一组值。此外，因为引用（即该数组的索引）为数字，我们可以进一步抽象。将引用看做是某个存储位置的无解释集合 $\mathcal{L}$ 的元素，然后将存储看做是位置 l 到值的一个部分函数。用元变量 μ 表示存储。那么，一个引用就是一个位置，一个到存储的抽象索引。从现在起我们将用位置而不用引用或指针来强调这个抽象质量[①]。

下面，需要扩展操作语义，将存储考虑进来。因为求值一个表达式的结果通常将依赖于求值所在的存储内容，求值规则不应该只将一个项作为参数，还应该将一个存储作为参数。此外，因为一个项的求值可以引起存储上的副作用，这可能会影响其他项在以后的求值，所以求值规则需要送回一个新的存储。这样，单步求值关系的形式从 $t \to t'$ 改变为 $t \mid \mu \to t' \mid \mu'$，其中 μ

① 这样抽象处理位置将防止我们对在像 C 这样的低级语言中的运行指针算术建模。这个限制是故意的。尽管指针算术有时是很有用的（特别在实现执行时间系统的低级部分，如垃圾收集时），大部分类型系统不能采用指针算术：在存储中位置 n 包含一个 Float 并不能告诉我们位置 $n+4$ 的类型。在 C 语言中，指针算术是违反类型安全性的一个主要因素。

和 μ'是存储的开始和结束状态。实际上,我们已经丰富了抽象机的概念,使得一个机器状态不只是一个程序计数器(表示为一个项),而是一个程序计数器加上存储的当前内容。

为实现这个改变,我们首先需要扩充现有的求值规则,使其包含存储因素:

$$(\lambda x{:}T_{11}.t_{12})\ v_2 \mid \mu \longrightarrow [x \mapsto v_2]t_{12} \mid \mu \qquad \text{(E-Appabs)}$$

$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{t_1\ t_2 \mid \mu \longrightarrow t_1'\ t_2 \mid \mu'} \qquad \text{(E-App1)}$$

$$\frac{t_2 \mid \mu \longrightarrow t_2' \mid \mu'}{v_1\ t_2 \mid \mu \longrightarrow v_1\ t_2' \mid \mu'} \qquad \text{(E-App2)}$$

注意这里的第一个规则返回不变的存储 μ:函数应用本身没有副作用。其他两个规则简单地将副作用从前提传播到结论。

下面,我们将对项的语法做一个小的改动。求值一个 ref 表达式的结果将是一个新位置,这样就需要将位置包括求值结果集中——即在值的集合中:

v ::=	值:
λx:T.t	抽象值
unit	单位值
l	存储位置

因为所有的值也是项,这意味着项的集合应该包括位置:

t ::=	项:
x	变量
λx:T.t	抽象
t t	应用
unit	常量unit
ref t	引用创建
!t	反引用
t:=t	赋值
l	存储位置

当然,将这个扩展到项的语法上并不意味着我们打算让程序员写带有明显具体位置的项:这样的项只作为求值的中间结果。事实上,在这一章中的项语言应该认为是对中间语言的形式化,它的某些特征对程序员来说不能直接使用。

借助于这个扩张的语法,我们可以陈述关于新的处理位置和存储构造的求值规则。首先,为了求值一个反引用的表达式!t_1,必须先归约 t_1 直到它变成一个值:

$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{!t_1 \mid \mu \longrightarrow !t_1' \mid \mu'} \qquad \text{(E-Deref)}$$

一旦完成 t_1 的归约,应该得到一个形为!l 的表达式,其中 l 是某个位置。一个企图对其他类的项值进行反引用,如对一个函数或 unit 都是错误的。求值规则在这种情况下将出现受阻现象。在 13.5 节中的类型安全性性质确保良类型项将不会出现这样的行为:

$$\frac{\mu(l) = v}{!l \mid \mu \longrightarrow v \mid \mu} \qquad \text{(E-DerefLoc)}$$

下面,为了求值一个赋值表达式 $t_1 := t_2$,必须先求值 t_1 直到它变成一个值(即一个位置):

$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{t_1 := t_2 \mid \mu \longrightarrow t_1' := t_2 \mid \mu'} \qquad \text{(E-Assign1)}$$

然后求值 t_2 直到它变成一个(任何种类的)值:

$$\frac{t_2 \mid \mu \longrightarrow t_2' \mid \mu'}{v_1 := t_2 \mid \mu \longrightarrow v_1 := t_2' \mid \mu'} \qquad \text{(E-Assign2)}$$

一旦完成了对 t_1 和 t_2 的求值,就得到一个形为 $l := v_2$ 的表达式,我们执行它来更新存储使位置 l 包含 v_2:

$$l := v_2 \mid \mu \longrightarrow \mathtt{unit} \mid [l \mapsto v_2]\mu \qquad \text{(E-Assign)}$$

(这里的记号$[l \mapsto v_2]\mu$表示"将 l 映射为 v_2 和对所有其他位置的映射为本身,如 μ 的存储"。注意,由求值得到的项是 unit;有趣的结果是更新的存储)。

最后,为了对形为 ref t_1 的一个表达式求值,首先求值 t_1 直到它变成一个值:

$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{\mathtt{ref}\ t_1 \mid \mu \longrightarrow \mathtt{ref}\ t_1' \mid \mu'} \qquad \text{(E-Ref)}$$

然后,为了求值 ref 本身,选择一个新位置 l(即不是 μ 定义域的部分中位置)并且产生一个存储用绑定 $l \mapsto v_1$ 来扩展 μ:

$$\frac{l \notin dom(\mu)}{\mathtt{ref}\ v_1 \mid \mu \longrightarrow l \mid (\mu, l \mapsto v_1)} \qquad \text{(E-Refv)}$$

由这一步得到的项是新近分配位置的名称 l。

注意这些求值规则不执行任何垃圾收集:我们允许存储随着求值进行无限制地增加。这不影响求值结果的正确性(毕竟,"垃圾"指的是不再可达,因此在求值过程中不再起任何作用的存储部分)。但它意味着求值器的一次无知的执行,将有时会用尽内存,而这时一个更复杂的求值器将重用那些内容变成垃圾的位置。

13.3.1 练习[★★★]:为了对垃圾收集建模,如何改进求值规则?为了说明这样的改进是正确的,我们需要证明什么样的定理?

13.4 存储类型

将语法和求值规则加入引用以后,最后的任务是写出新的构造类型规则,并且,检查它们是可靠的。自然地,关键问题是"一个位置的类型是什么?"

当我们求值一个含有具体位置的项时,结果的类型依赖于开始存储的内容。比如,如果我们在存储$(l_1 \mapsto \mathtt{unit}, l_2 \mapsto \mathtt{unit})$中求值项$!l_2$,结果是 unit;如果在存储$(l_1 \mapsto \mathtt{unit}, l_2 \mapsto \lambda x{:}\mathtt{Unit}.x)$中求值相同的项,结果是 $\lambda x{:}\mathtt{Unit}.x$。关于前一个存储,位置 l_2 有类型 Unit,而后者有类型 Unit→Unit。这使我们立即想出如下关于位置的一个类型规则:

$$\frac{\Gamma \vdash \mu(l) : T_1}{\Gamma \vdash l : \mathtt{Ref}\ T_1}$$

即为找一个位置 l 的类型,在存储中查找 l 的当前内容,并计算内容的类型 T_1。位置的类型就是 Ref T_1。

如果是这样,就需要进一步考虑才能达到一个协调的状态。事实上,使一个项的类型依赖

于存储时，我们已经改变了类型关系，从一个三元关系(上下文、项和类型之间)变成一个四元关系(上下文、存储、项和类型之间)。因为存储直觉上是我们计算一个项的类型所在上下文的一个部分，所以可将这个带存储的四元关系为 $\Gamma|\mu \vdash t:T$。引用类型化规则现在的形式为：

$$\frac{\Gamma \mid \mu \vdash \mu(l) : T_1}{\Gamma \mid \mu \vdash l : \text{Ref } T_1}$$

并且系统中其余类型规则可以类似于存储进行扩展。其他规则与存储没有任何有趣的关系，只是直接将存储从前提传播到结论。

然而，这个规则仍有两个问题。首先，类型检查效率不高，因为计算一个位置 l 的类型包括计算当前 l 内容 v 的类型。如果 l 在一个项 t 中出现许多次，我们将在构造 t 的一个类型推导中计算 v 的类型许多次。更糟的是，如果 v 本身包含位置，则将不得不在它们出现的每一次再次计算它们的类型。比如，如果存储包含：

```
(l1 ↦ λx:Nat. 999,
 l2 ↦ λx:Nat. (!l1) x,
 l3 ↦ λx:Nat. (!l2) x,
 l4 ↦ λx:Nat. (!l3) x,
 l5 ↦ λx:Nat. (!l4) x),
```

则计算 l_5 的类型包括计算 l_4，l_3，l_2 和 l_1 的类型。

其次，上面提到的位置类型规则可能在存储包含一个循环时推导不出任何结果。比如，关于存储：

```
(l1 ↦ λx:Nat. (!l2) x,
 l2 ↦ λx:Nat. (!l1) x),
```

位置 l_2 没有有限的推导，因为计算 l_2 的一个类型要求找到 l_1 的类型，它反过来又包含 l_2 等，循环引用结构在实际中常常出现(比如，可以用来建立双链接列表)，并且希望我们的类型系统能够处理它们。

13.4.1　练习[★]：能否找到一个项，它的求值将产生这个特别的循环存储？

这两个问题都是由下面的事实引起的：上面提到的位置类型规则要求每次在一个项中提到位置时再次计算它的类型。但直觉上没有必要这样做。毕竟，当一个位置第一次创建时，我们知道存储到它里面的初始值的类型。此外，尽管以后可能存储其他值到这个位置，那些值将总是与初始值的类型相同。换言之，我们头脑中对存储中的每个位置总是有一单个的确定类型，当位置被分配时就固定了。这些想要的类型可以收集在一起，作为一个存储类型——一个将位置映射到类型的有限函数。我们将用元变量 Σ 表示这样的函数。

假设用存储类型 Σ 来描述将被求值的某项 t 所在的存储 μ。则能用 Σ 来计算 t 的结果类型，而不需要直接根据 μ。比如，如果 Σ 是($l_1 \mapsto$ Unit，$l_2 \mapsto$ Unit→Unit)，则可以直接推导出 $!l_2$ 有类型 Unit→Unit。更一般地，位置类型规则可以根据：

$$\frac{\Sigma(l) = T_1}{\Gamma \mid \Sigma \vdash l : \text{Ref } T_1} \qquad \text{(T-Loc)}$$

存储类型规则来重新形式化。一个四元关系也可以类型化，但它是对一个存储类型参数化而不是对一个具体存储。其余的类型规则可以类似地用存储类型来扩充。

当然,只有在求值过程中用到的具体存储确实一致于我们为了类型检查而假定的存储类型时,这些类型规则才将精确预测求值的结果。这个附带条件正好平行于到目前为止所见演算中含自由变量的情况:根据代换引理[参见引理(9.3.8)]有如果 $\Gamma \vdash t{:}T$ 则可以用列在 Γ 中类型的值替代 t 中的自由变量,而得到一个类型 T 的封闭项,由类型保持定理[参见定理(9.3.9)],这个封闭项将求值为类型 T 的最后结果(如果它产生任何结果的话)。我们在 13.5 节中将看到如何形式化一个类似于存储和存储类型的概念。

最后要注意,为了对程序员实际写的项进行类型检查,不需要费任何周折来猜测我们用的是什么存储类型。如上面所述,具体位置常量只出现在作为求值中间结果的项中;它们不会出现在程序员写程序用的语言中。这样,可以根据空存储类型来类型检查程序员的项。随着求值的进行和新位置的创建,将总是能够看到如何着眼于新的分配单元中初始值的类型来扩展存储类型;这个直觉在下面的类型保持定理[参见定理(13.5.3)]的陈述中形式化。

既然解决了位置问题,其他新的语法形式的类型规则马上就可以得出。当我们产生一个引用到类型 T_1 的一个值时,引用本身有类型 Ref T_1:

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{t}_1 : \mathtt{T}_1}{\Gamma \mid \Sigma \vdash \mathtt{ref}\ \mathtt{t}_1 : \mathtt{Ref}\ \mathtt{T}_1} \qquad \text{(T-Ref)}$$

注意,这里不需要扩展存储类型,因为新位置的名称在执行时间之前是不会确定的,而 Σ 只记录已经分配的存储单元和它们类型之间的关联。

反之,如果 t_1 求值结果类型为 Ref T_{11} 的一个位置,则对 t_1 仅引用将确保产生类型为 T_{11} 的一个值:

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{t}_1 : \mathtt{Ref}\ \mathtt{T}_{11}}{\Gamma \mid \Sigma \vdash\ !\mathtt{t}_1 : \mathtt{T}_{11}} \qquad \text{(T-Deref)}$$

最后,如果t_1表示类型 Ref T_{11} 的一个单元,则我们能存储t_2到这个单元中,只要t_2的类型也是 T_{11}:

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{t}_1 : \mathtt{Ref}\ \mathtt{T}_{11} \qquad \Gamma \mid \Sigma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \mid \Sigma \vdash \mathtt{t}_1 {:=} \mathtt{t}_2 : \mathtt{Unit}} \qquad \text{(T-Assign)}$$

图 13.1 总结了带引用的简单类型 lambda 演算的类型规则(及为方便引用提出的语法和求值规则)。

13.5 安全性

本章的最后任务是检查含引用的演算是否仍然符合标准类型安全性质。进展定理(良类型项不会受阻)[参见定理(13.5.7)]的提出与证明大致与前几章相似;我们只需要加上几个直接可证的情况来处理新的构造。保持定理有点意思,因此我们首先来看看保持定理。

因为扩展了求值关系(带初始和最终存储)和类型关系(带一个存储类型),我们需要改变保持的语句将这些参数包括进来。显然,不能只加上存储和存储类型而不考虑它们之间的联系。

$$\text{If } \Gamma \mid \Sigma \vdash \mathtt{t} : \mathtt{T} \text{ and } \mathtt{t} \mid \mu \longrightarrow \mathtt{t}' \mid \mu', \text{ then } \Gamma \mid \Sigma \vdash \mathtt{t}' : \mathtt{T}. \qquad \textbf{(错误!)}$$

如果根据存储值类型的假设集合进行类型检查,并且对违反这些假设的一个存储来求值,结果将是一个灾难。下面提出了必要的约束。

13.5.1 定义: 对于一个存储 μ,一个类型上下文 Γ 和一个存储类型 Σ,如果 $dom(\mu) = dom(\Sigma)$ 且对每个 $l \in dom(\mu)$,$\Gamma|\Sigma \vdash \mu{:}\Sigma(l)$成立,则称 μ 为良类型的,记为 $\Gamma|\Sigma \vdash \mu(l){:}\Sigma(l)$。

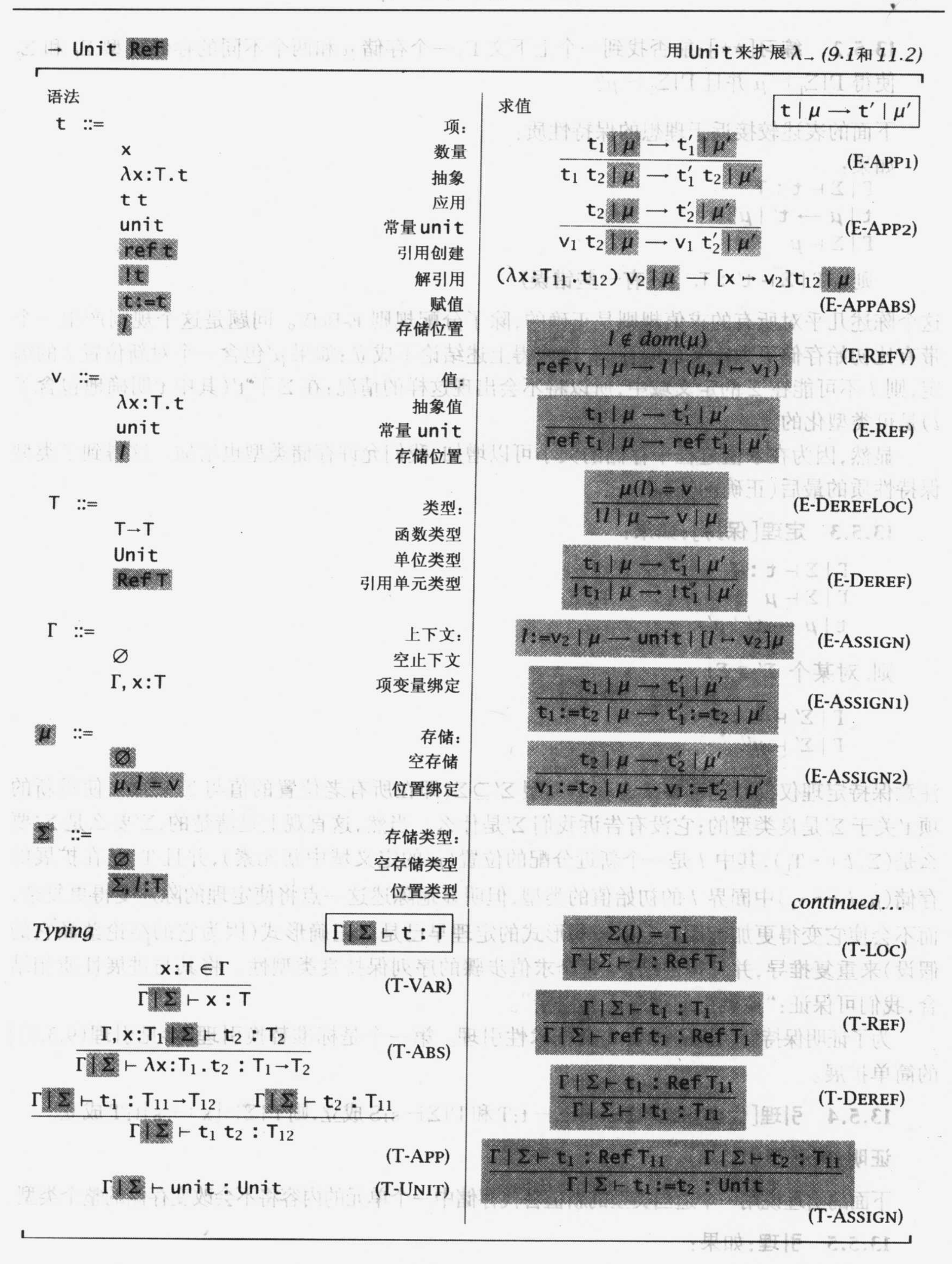


图 13.1 引用

直觉地，如果在存储中每个值具有存储类型预测的类型，一个存储 μ 与一个存储类型 Σ 是一致的。

13.5.2 练习[★★]:能否找到一个上下文 Γ,一个存储 μ 和两个不同的存储类型 Σ_1 和 Σ_2 使得 $\Gamma|\Sigma_1 \vdash \mu$ 并且 $\Gamma|\Sigma_2 \vdash \mu$?

下面的表述较接近于理想的保持性质:

如果:

$\Gamma \mid \Sigma \vdash \mathsf{t} : \mathsf{T}$

$\mathsf{t} \mid \mu \longrightarrow \mathsf{t}' \mid \mu'$

$\Gamma \mid \Sigma \vdash \mu$

则, $\Gamma \mid \Sigma \vdash \mathsf{t}' : \mathsf{T}$. (还有一些错误)

这个陈述几乎对所有的求值规则是正确的,除了分配规则 E-RefV。问题是这个规则产生一个带有比初始存储更大定义域的存储,这使得上述结论不成立:如果 μ'包含一个对新位置 l 的绑定,则 l 不可能在 Σ 的定义域中,所以将不会出现这样的情况:在 Σ 下 t′(其中 t′明确地包含了 l)是可类型化的。

显然,因为在求值过程中存储的大小可以增加,我们允许存储类型也增加。这得到了类型保持性质的最后(正确的)陈述。

13.5.3 定理[保持]:如果:

$\Gamma \mid \Sigma \vdash \mathsf{t} : \mathsf{T}$

$\Gamma \mid \Sigma \vdash \mu$

$\mathsf{t} \mid \mu \longrightarrow \mathsf{t}' \mid \mu'$

则,对某个 $\Sigma' \supseteq \Sigma$:

$\Gamma \mid \Sigma' \vdash \mathsf{t}' : \mathsf{T}$

$\Gamma \mid \Sigma' \vdash \mu'$.

注意保持定理仅仅断定存在某个存储类型 $\Sigma' \supseteq \Sigma$(即在所有老位置的值与 Σ 一致),使得新的项 t′关于 Σ'是良类型的;它没有告诉我们 Σ'是什么。当然,这直观上是清楚的,Σ'要么是 Σ,要么是$(\Sigma, l \mapsto \mathrm{T}_1)$,其中 l 是一个新近分配的位置(μ'的定义域中新元素),并且 T_1 是在扩展的存储$(\mu, l \mapsto \mathrm{v}_1)$中囿界 l 的初始值的类型,但明显地陈述这一点将使定理的陈述变得更复杂,而不会使它变得更加有用。上面较弱形式的定理早已是以正确形式(因为它的结论蕴涵它的假设)来重复推导,并且得出结论:每个求值步骤的序列保持良类型性。将其与进展性质相结合,我们可保证:“良类型的程序不会出错”。

为了证明保持性,我们需要几个技术性引理。第一个是标准替换引理[参见引理(9.3.8)]的简单扩展。

13.5.4 引理[代换]:如果 $\Gamma,\mathrm{x}:\mathrm{S}|\Sigma \vdash \mathrm{t}:\mathrm{T}$ 和 $\Gamma|\Sigma \vdash \mathrm{s}:\mathrm{S}$ 成立,则 $\Gamma|\Sigma \vdash [\mathrm{x} \mapsto \mathrm{s}]\mathrm{t}:\mathrm{T}$ 成立。

证明:同引理(9.3.8)。

下面的引理说用一个适当类型的新值替代存储中一个单元的内容将不会改变存储的整个类型。

13.5.5 引理:如果:

$\Gamma \mid \Sigma \vdash \mu$

$\Sigma(l) = \mathsf{T}$

$\Gamma \mid \Sigma \vdash \mathsf{v} : \mathsf{T}$

则, $\Gamma \mid \Sigma \vdash [l \mapsto \mathsf{v}]\mu$

证明：直接由 $\Gamma|\Sigma\vdash\mu$ 的定义得到。

最后，我们需要一种存储的弱化引理，如果一个存储用一个新的位置扩展，则扩展的存储仍允许我们给以前相同的项指派类型。

13.5.6 引理：如果 $\Gamma|\Sigma\vdash t:T$ 成立且 $\Sigma'\supseteq\Sigma$，则有 $\Gamma|\Sigma'\vdash t:T$。

证明：简单归纳证明。

现在我们证明主要的保持定理。

保持定理(13.5.3)：直接对求值推导做归纳，利用上面的引理和类型规则的逆转性质[引理(9.3.1)的一个直接推广]。

对进展定理[参见定理(9.3.5)]的扩展必须要考虑存储及存储类型。

13.5.7 定理[进展]：假设 t 是一个封闭的、良类型的项(即对某个 T 和 Σ，有 $\varnothing|\Sigma\vdash t:T$)。则要么 t 是一个值，要么对任何存储 μ 使 $\varnothing|\Sigma\vdash\mu$，存在某个项 t′和存储 μ' 使得 $t|\mu\rightarrow t'|\mu'$。

证明：直接对类型推导做归纳，如定理(9.3.5)的形式[典型形式引理(9.3.4)需要两个附加的情况，用来说明所有类型 Ref T 的值是位置，与 Unit 类似]。

13.5.8 练习[推荐，★★★]：本章中的求值关系是否在良类型的项上是规范化的？如果是，证明它；如果不是，在当前的演算(带有数值和布尔值)中写出一个良类型的阶乘函数。

13.6 注释

本章中的内容是根据Harper(1994，1996)的处理方式修改而成的。Wright和Felleisen(1994)给出了一个风格类似的说明。

将引用(或其他计算效果)组合到 ML 形式的多态类型推论中产生了某些相当麻烦的问题(参见 22.7 节)，且这种组合在研究文献中受到重视。参见Tofte(1990)，Hoang 等(1993)，Jouvelot 和Gifford(1991)，Talpin和Jouvelot(1992)，Leroy和Weis(1991)，Wright(1992)，Harper(1994，1996)，以及他们所列的参考文献。

别名的静态预测是一个在编译实现(称为别名分析)和程序语言理论中长期未解决的问题。Reynolds(1978，1989)的一个有影响的尝试被冠名为干扰的语法控制。这些想法最近引发一个新活动——参见O'Hearn 等(1995)和 Smith 等(2000)。Reynolds(1981)以及Ishtiaq和O'Hearn(2001)所列的参考文献中讨论了对别名的更一般的推理技术。

在Jones和Lins(1996)中可以找到关于垃圾收集的一个全面论述。更偏重于语义的处理在Morrisett 等(1995)中有所论述。

> 我们现在就应找出致使他发疯的原因，或令其发疯的某些缺陷，因为疯症是个结果，而此结果必定是某缺陷所造成的。
>
> ——哈姆雷特 II，第 2 幕

> 指向月亮的手指不是月亮。
>
> ——佛说

第 14 章　异　常[①]

在第 13 章中，我们看到了如何扩展带可变引用的纯简单类型 lambda 演算的简单操作语义，并考虑了这种扩展对类型规则和类型安全性证明的影响。在这一章中，我们处理另一种计算模型的扩展：提升和处理异常。

实际编程中经常会遇到这种情况：一个函数需要向它的调用者发出一个信息，由于某种原因不能执行它的任务，因为某个计算包含被零整除，或一个算术溢出，一个查找键从字典中的丢失，数组下标越界，一个文件不能找到或打开，以及某个灾难性事件的发生，如系统内存满了，或用户强行终止了某个进程，等等。

这些异常条件中某一些可以通过使函数返回一个变式（或一个选择）来显示，如在 11.10 节中所见到的那样。但在异常条件是真正的异常情形中，我们不可以要求函数每次来处理可能出现的异常，而更希望一个异常条件引起一个直接控制转换为程序更高层中定义异常处理器（如果异常条件是非常罕见的，或如果调用者没有办法从异常中恢复）简单地终止程序。我们首先考虑后一种情况（参见 14.1 节），其中的一个异常是放弃整个程序，然后加上一个机制来捕获异常并恢复（参见 14.2 节），最后改进这两个机制，允许只有程序员说明的数据可以在异常的点和处理器之间传递（参见 14.3 节）。

14.1　提升异常

在简单类型 lambda 演算中加入可能最简单的提示异常的机制：一个项 error，当求值时，如果该项出现，则完全终止项的求值。图 14.1 给出了所需要的扩展。

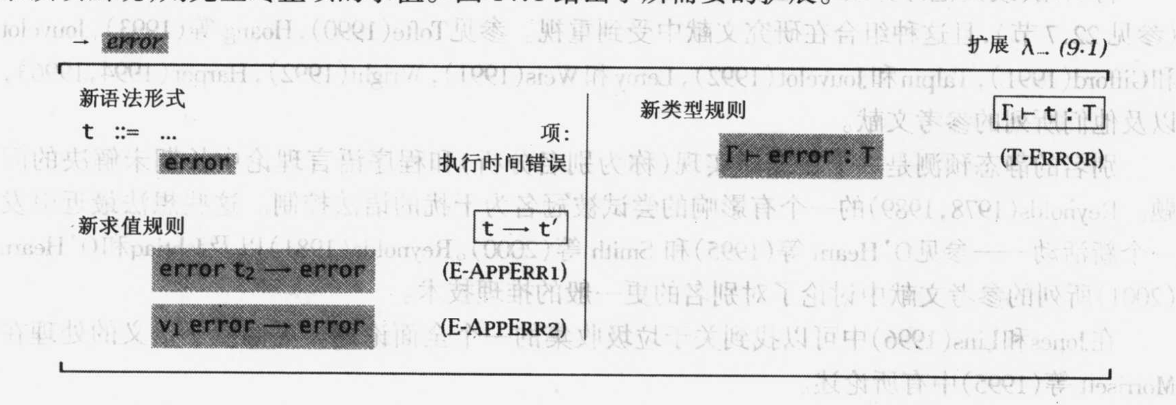


图 14.1　错误

在写 error 的规则时，最主要考虑的是在操作语义中如何形式化“非正常终止”。我们简单采用让 error 本身就是放弃一个程序的结果。用规则 E-AppErr1 和规则 E-AppErr2 说明这个做

① 本章中讨论的系统是扩展为包含各种异常和异常处理原语（如图 14.1 和图 14.2 所示）的简单类型 lambda 演算（参见图 9.1）。第一个扩展的 OCaml 实现 fullerror。带值异常（如图 14.3 所示）的语言没有实现。

法。E-AppErr1 说明:如果将应用的左端归约为一个值时遇到项 error,我们应该立即产生 error 作为应用的结果。类似地,E-AppErr2 说明:如果将应用的参数归约为一个值时遇到一个 error,则应该放弃应用并立即产生 error。

注意,我们没有将 error 包括在值的语法中,只是在项的语法中。这保证了在 E-AppAbs 和 E-AppErr2 规则的左端永远不会出现重叠,即求值:

```
(λx:Nat.0) error
```

这样的一个项不会含糊不清:是执行应用(产生 0 作为结果)还是放弃。结果是选择后者。类似地,在 E-AppErr2 中使用元变量 v_1(而不是取任何项 t_1)迫使求值器等待,直到一个应用的左端在放弃之前归约为一个值,即使右端是 error。这样,如:

```
(fix (λx:Nat.x)) error
```

这样的项将发散而不是终止。这些条件确保求值关系仍然是可确定的。

类型规则 T-Error 也很有意思。因为我们允许在任何上下文中提升一个异常,项 error 的形式可以有任何的类型。在:

```
(λx:Bool.x) error;
```

中它有类型 Bool。在:

```
(λx:Bool.x) (error true);
```

中它有类型 Bool→Bool 。

error 类型的这个灵活性在实现类型检查算法时会提高难度,因为它破坏了下面的性质:每个在语言中可类型化的项有惟一的类型[参见定理(9.3.3)]。这能用几种方法来处理。在带有子类型的语言中,我们能给 error 赋予最小类型 Bot(参见 15.4 节),它在必要时可以提升为任何其他类型。在带参数多态的语言中(参见第 23 章),能给 error 多态类型 $\forall X.X$,它能实例化为任何其他类型。这两种办法允许 error 的无限多个可能的类型紧致地表示为一个单一类型。

14.1.1 练习[★]:如果要求程序员在使用 error 的每个上下文中用合适的类型注释error,这样是否更简单?

对于带异常语言的类型保持性质也成立:如果一个项有类型 T,并且我们对它一步求值,结果仍为类型 T。进展性质则需要做点修改。在原来的形式中,进展性质说明一个良类型程序必须求值为一个值(或发散)。但现在我们引入了一个非值的范式 error,它可以是一个良类型程序的求值结果。根据这一点需要重述进展。

14.1.2 定理[进展]:假设 t 是一个封闭的良类型范式,则要么 t 是一个值,要么t = error。

14.2 处理异常

error 的求值规则可以看做是"展开调用堆栈",放弃未决的函数调用直到 error 扩散到顶层。在实际带异常语言的实现中,会发生:调用堆栈由活动记录的集合组成,每个活跃函数调用有一个活动记录;提升一个异常引起活动记录从调用堆栈中弹出直到堆栈变空为止。

在大部分带异常的语言中，也可能在调用堆栈中装上一个异常处理器。当一个异常提升时，活动记录从调用堆栈中弹出直到遇到一个异常处理器，并且求值在有这个处理器的情况下继续进行。换言之，异常函数作为控制的一个非局部转换器，它的目标是最近装上的异常处理器（即在调用堆栈中最近的一个）。

对异常处理器的形式化类似于 ML 和 Java（参见图 14.2）。表达式 try t_1 with t_2 表示"送回 t_1 的求值结果，除非它放弃，放弃时会求值处理器 t_2"。求值规则 E-TryV 说明：当 t_1 归约为一个值 v_1 时，可以丢掉 try，因为它不会再需要了。另一方面，E-TryError 说明：如果求值 t_1 导致 error，则可以用 t_2 替代 try，并且从那里继续求值。E-Try 告诉我们在归约 t_1 为一个值或 error 之前，应该继续对 t_1 求值并且考虑 t_2。

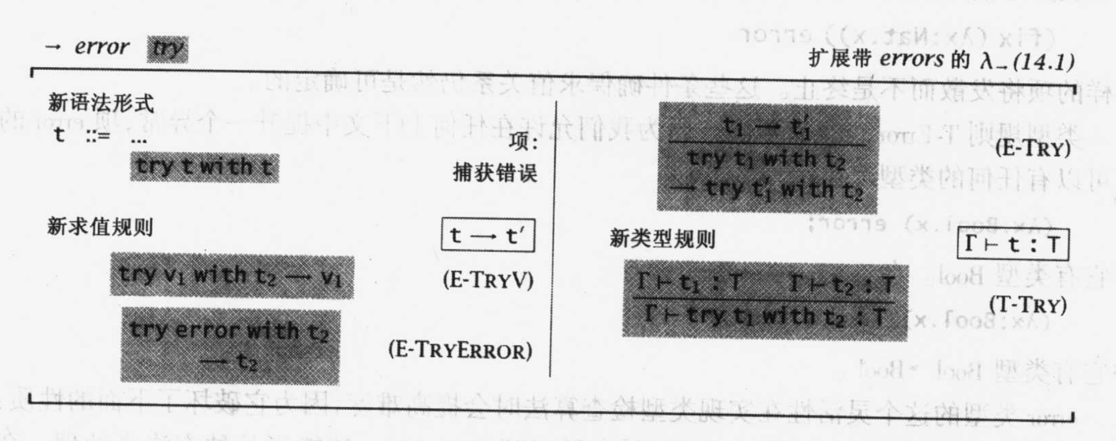


图 14.2 error 处理

try 的类型规则直接根据它的操作语义。整个 try 的结果可以是主体 t_1 的结果，也可以是处理器 t_2 的结果；如果它们有相同的类型 T，try 的类型也为 T。

类型安全性性质和它的证明本质上与上一节相同。

14.3 带值的异常

在 14.1 节和 14.2 节中引入的机制使函数提示它的调用者"某种不寻常的事情发生了"。提示发生了哪些不寻常的事，这种信息是很有意义的，因为处理器需要采取的行动（要么恢复并且重试，要么将一个可理解的错误消息反馈给用户）将依赖于这个信息。

图 14.3 说明基本异常处理的构造如何可以加强，使得每个异常携带一个值。这个值的类型记为 T_{exn}。暂时我们不设定这个类型的确切性质；下面将讨论几种可能的选择。

原子项 error 用一个项构造子 raise t 替代，其中 t 是一个我们需要传给异常处理器的额外信息。try 的语法仍然不变，但 try t_1 with t_2 中的处理器 t_2 现在解释为一个函数，它取这个额外信息为参数。

求值规则 E-TryRaise 实现这个行为，从体 t_1 中取出由一个 raise 携带的额外信息，将它传给处理器 t_2。E-AppRaise1 和 E-AppRaise2 在应用中传递异常，就像图 14.1 中的 E-AppErr1 和 E-AppErr2。然而要注意，这些规则只允许传递额外信息为一个值的异常；如果我们企图求值一个带额外信息的 raise，而额外信息本身要求求值，这些规则将受阻，迫使我们先采用 E-Raise

来对额外信息求值。当求值的额外信息是其他异常中发送的时候会发生异常,而这种异常由 E-RaiseRaise 传递。E-TryV 告诉我们,一旦一个 try 的主体归约为一个值就可以丢弃 try,就像在 14.2 节中一样。E-Try 引导求值器在一个 try 的体上求值直到它成为一个值或一个 raise。

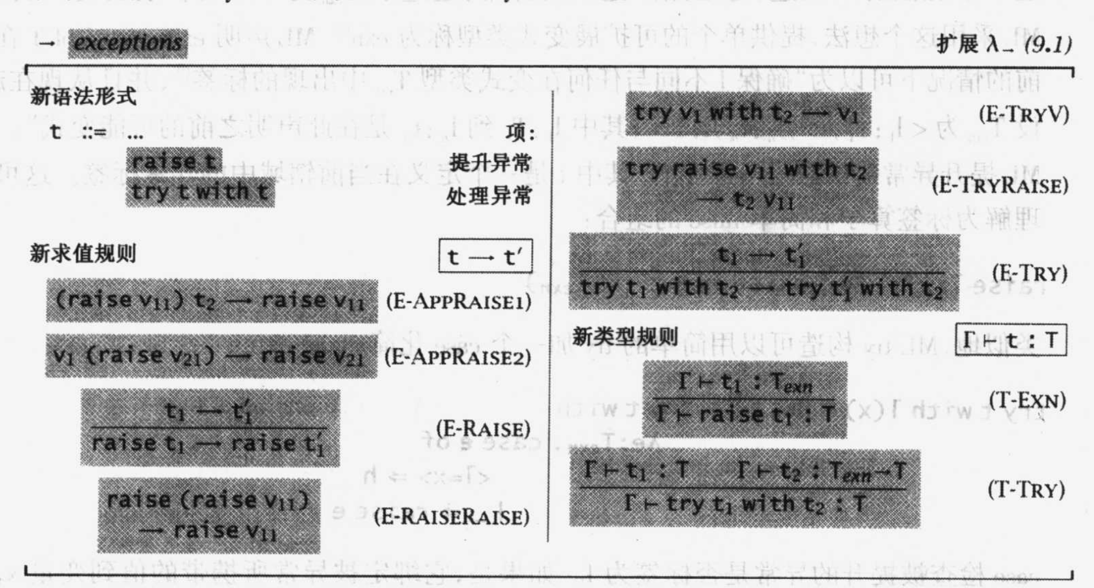


图 14.3 带值的异常

行为上,类型规则反映这些改变。在 T-Raise 中我们要求额外信息有类型 T_{exn};整个 raise 可以赋予上下文要求的任何类型 T。在 T-Try 中,把处理器 t_2 当成一个函数,对给定类型 T_{exn} 的额外信息,产生与 t_1 相同类型的一个结果。

最后,考虑类型 T_{exn} 的可能方案。

1. 可以取 T_{exn} 为 Nat 。这对应于 UNIX 操作系统函数所用的 errno 约定:每个系统调用返回一个数值的"错误代码",用 0 表示成功,其他值表示各种异常条件。
2. 可以取 T_{exn} 为 String,这可以避免在表中查询错误数,并且允许异常提升点构造所希望的更具描述性的消息。这个额外灵活性的代价是错误处理器可能不得不分析这些字符串来找出发生了什么。
3. 可以继续将信息更多的异常传递出去,而避免字符串分析,条件是定义 T_{exn} 为一个变式类型:

```
Texn   =   <divideByZero:     Unit,
            overflow:         Unit,
            fileNotFound:     String,
            fileNotReadable:  String,
            ...>
```

   这个方案允许一个处理器用一个简单 case 表达式来区别各种异常。不同的异常可以携带不同类型的附加信息:像 divideByZero 这样的异常不需要额外打包, fileNotFound 可以携带一个字符串标示在错误出现时哪个文件被打开过,等等。

但这个方案有点不灵活，需要事先固定能由程序提升的异常整个集合(即变式类型 T_{exn} 的标签的集合)。这样程序员没有余地来声明应用特有的异常。

4. 上一个想法可以改进，考虑用户定义的异常，通过取 T_{exn} 为一个可扩展的变式类型。ML 采用这个想法，提供单个的可扩展变式类型称为 exn[1] ML 声明 exception l of T 在当前的情况下可以为"确保 l 不同与任何在变式类型 T_{exn} 中出现的标签[2]，并且从现在起，设 T_{exn} 为 $<l_1:T_1,\cdots,l_n:t_n,l:T>$，其中 $l_1:T_1$ 到 $l_n:t_n$ 是在此声明之前的可能变式"。

   ML 提升异常的语法是 raise l(t)，其中 l 是一个定义在当前辖域中的异常标签。这可以理解为标签算子和简单 raise 的组合：

```
raise l(t)   ≝   raise (<l=t> as Texn)
```

   类似地，ML try 构造可以用简单的 try 加一个 case 化简：

```
try t with l(x) → h   ≝   try t with
                               λe:Texn. case e of
                                          <l=x> ⇒ h
                                        | _ ⇒ raise e
```

   case 检查被提升的异常是否标签为 l。如果是，它绑定被异常所携带的值到变量 x，并且对处理器 h 求值。如果不是，它到 else 子句，重新提升异常。异常将继续传递(并可能被捕获和重新提升)直到它要么到达一个要求处理它的处理器，要么到达顶层并且放弃整个程序。

5. Java 用类(而不是可扩展的变式)来支持用户定义的异常。这个语言提供一个内置类 Throwable；Throwable 的一个实例或它的子类实例可以在一个 throw 中(如同 raise)，或 try…catch(如同 try…with)语句中使用。新的异常可以通过定义 Throwable 的新的子类来声明。

   实际上这个异常处理机制和 ML 的机制有密切的对应。粗略地讲，在 Java 中一个异常对象在运行时间中表示为一个指示它的类的一个标签(这直接对应于在 ML 中可扩展的变式标签)，加上一个实例变量的记录(对应于由这个标签标记的额外信息)。

   Java 异常在许多方面比 ML 更进一步。其中一个是在异常标签上存在一个自然偏序，这个偏序是由子类序生成的。异常 l 的一个处理器实际上捕捉携带一个类 l 或任何 l 子类对象的所有异常。另一个是 Java 区分应用程序想要捕捉并试图恢复的异常(内置类 Exception 的子类—— Throwable 的一个子类)和指示一个通常应该停止执行的严重状况的错误(Error 的子类——也是 Throwable 的一个子类)，两者之间的关键区别在于类型检查规则，这些规则要求方法明确地声明它们提升的是哪个异常(而不是哪个错误)。

14.3.1 练习[★★★]：在上述的方案4中可扩展的变式类型的解释有点非形式化。说明如何使它精确化？

---

① 可以进一步地将可扩展的变式类型作为一个一般的语言特征，但 ML 的设计者选择将 exn 处理为一个特殊情况。

② 因为 exception 的形式是一个绑定器，我们总可以保证 l 不同于在 T_{exn} 中使用的标签，如有必要可利用 alpha 转化。

14.3.2　练习[★★★★]:注意到 Java 异常(那些 Exception 的子类)比起 ML 中异常(或这里定义的)限制更多:由方法可以提升的异常必须在方法的类型中声明。扩展你对练习 14.3.1 的解答,使得一个函数的类型指示不但是它的参数和结果类型,而且也指示它可能提升的异常集合。证明你的系统是类型安全的。

14.3.3　练习[★★★]:许多其他控制构造子可以用类似于在本章中见到的技术来形式化。熟悉 Scheme 的"当前连续调用"(call/cc)算子(参见 Clinger,Friedman 和 Wand,1985; Kelsey,Clinger 和 Rees,1998; Dybvig,1996; Friedman,Wand 和 Haynes,2001)的读者,可以试着形式化基于 T 连续的一个类型 Cont T(即期望类型 T 的一个参数的连续)上的类型规则。

第三部分　子 类 型 化

第 15 章　子　类　型[①]

在第二部分中，我们用最后几个章节的篇幅学习了在简单类型的 λ 演算框架内的语言特征的种种类型化行为。本章将着重讲述一个更加基本的扩展部分：子类型化（有时也称为子类型的多态性）。到目前为止我们所讨论的内容之间的联系并不紧密，与之不同的是，子类型是一个交叉的领域，对绝大多数的语言特征之间有着非常重要的影响。

子类型化是面向对象程序语言所特有的，并且通常认为是面向对象类型的一个最本质的特征。在第 18 章将对这方面的内容做进一步探讨；现在，尽管已经出现了很多有趣的课题，但我们对子类型化的表述仍将停留在简单的形式中，即只讨论函数和记录。在 15.5 节中，将讨论子类型与前几章提到的其他特征的结合。在 15.6 节将用更加精确的语言描述子类型，并与运行时间强制转换的加入对应。

15.1　包含

如果没有子类型，简单类型的 λ 演算规则的实现会遇到很大的困难。系统类型的存在，使参数类型完全与函数类型的字段相匹配，这将导致类型检查器抛弃众多的对程序员来讲似乎是有着明显良好性能的程序，比如，我们回顾一下作为函数应用的类型化规则：

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \qquad \text{(T-App)}$$

根据这个规则，正确的项：

```
(λr:{x:Nat}. r.x) {x=0,y=1}
```

是不能类型化的，因为参数类型是{x:Nat，y:Nat}，而函数所接受的参数却是{x:Nat}。但是，函数显然仅仅要求它的参数是带字段 x 的记录，它并不关心参数是否带其他字段。而且，能从这个函数的类型看出：我们并不需要查看函数体以确认除了字段 x 外它没有使用其他的字段。传递类型为{x:Nat，y:Nat}的参数给希望得到参数类型为{x:Nat}的函数总是安全的。

进行子类型化的目的是改进类型化规则，这样出现上面情况的项也是可接受的。一些类型比其他类型带来的信息更准确，为此要将该想法形式化：所说的 S 是 T 的子类型，记为 S <:T，意味着所有类型为 S 的项用在需要类型 T 的项的上下文中都是安全的。这个子类型化的观点通常被称为安全代换原则。

S <:T 更简单的理解是“用 S 刻画的每一个值就是用 T 来刻画的”，也就是说“S 型的元素集合是 T 型元素集合的子集”。我们在 15.6 节中将会看到其他对子类型化更加精确的解释有时是有用的，但是本书的子集语义足以满足大部分要求。

---

① 本章学习的演算是 $\lambda_{<:}$，它是带子类型化（参见图 15.1）和记录类型（参见图 15.3）的简单类型 λ 演算；相关的 OCaml 实现是 rcdsub（一些例子还使用了数字；应该用 fullsub 来检查它们）。

类型关系与这里的子类型关系之间的桥梁可通过增加一个称为包含的新类型规则建立起来：

$$\frac{\Gamma \vdash t : S \quad S <: T}{\Gamma \vdash t : T} \qquad \text{(T-Sub)}$$

这个规则告诉我们，如果 $S <: T$，那么 S 中的每一个元素也都是 T 中的元素。举例来说，如果定义子类型关系如 $\{x{:}Nat, y{:}Nat\} <: \{x{:}Nat\}$，那么就可以用公式(T-Sub)来推导出 $\vdash \{x=0, y=1\} : \{x{:}Nat\}$，这样提出的例子能通过类型检查。

15.2　子类型关系

子类型关系是将 $S <: T$[可读做"S 是 T 的一个子类型"(或"T 是 S 的父类型")]形式的推导语句组成的推论规则集成起来进一步形式化。我们分开考虑每种类型形式(如函数类型，记录类型等)；对每种类型，都将介绍一条或多条规则，用来将应该使用某个类型元素的地方却使用了另一种类型的元素而依然是安全的这种情况进行了形式化。

在展开特殊的类型构造子之前，先提出两个一般的约定：第一，子类型化应该有自反性：

$$S <: S \qquad \text{(S-Refl)}$$

第二，子类型化应该有传递性：

$$\frac{S <: U \quad U <: T}{S <: T} \qquad \text{(S-Trans)}$$

提出这些规则是直接出于安全代换的考虑。

现在，对于记录类型，有类型 $S = \{k_1 : S_1 \cdots k_m : S_m\}$ 和类型 $T = \{l_1 : T_1 \cdots l_n : T_n\}$，如果 T 含的字段数少于小于 S，就认为 S 是 T 的子类型。特别情况下，"忘记"记录类型最后的某些字段是无大碍的。这个思想可称为广度子类型化：

$$\{l_i : T_i \;^{i \in 1..n+k}\} <: \{l_i : T_i \;^{i \in 1..n}\} \qquad \text{(S-RcdWidth)}$$

这看上去有些不可思议：一个"更小"的类型(子类型)却有更多的字段。最简单的理解方法就是这里对记录类型采用一种比 11.8 节中用到的更为自由的观点，即将一个记录类型 $\{x{:}Nat\}$ 理解为"所有至少含 Nat 类型的字段 x 的记录集合。"值 $\{x=3\}$ 和 $\{x=5\}$ 就是这个类型的元素，同样，值 $\{x=3, y=100\}$ 和 $\{x=3, a=true, b=true\}$ 也是如此。同理，记录类型 $\{x{:}Nat, y{:}Nat\}$ 是描述至少含 x 和 y 两个字段，且 x 和 y 的类型都是 Nat 的记录。值 $\{x=3, y=100\}$ 和 $\{x=3, y=100, z=true\}$ 都是这个类型的成员，而 $\{x=3\}$ 则不是，$\{x=3, a=true, b=true\}$ 也不是。这样一来，属于第二个类型的值的集合就正好属于第一个类型值的集合的子集。记录的字段数越多限制也就越多，但带来的信息含量也越多，所以表示的值集就越小。

广度子类型化规则仅仅适用于公用字段完全相同的记录类型。只要两个记录中每个对应字段的类型仍是子类型关系，个别字段的类型改变也是允许的。这一点可用深度子类型化规则来说明：

$$\frac{\text{对每个 } i \quad S_i <: T_i}{\{l_i : S_i \;^{i \in 1..n}\} <: \{l_i : T_i \;^{i \in 1..n}\}} \qquad \text{(S-RcdDepth)}$$

接下来子类型化的推导是将 S-RcdWidth 和 S-RcdDepth 规则一起使用来说明嵌套的记录类型{x:{a:Nat,b:Nat},y:{m:Nat}}是类型{x:{a:Nat},y:{}}的子类型：

$$\dfrac{\dfrac{}{\mathtt{\{a:Nat,b:Nat\} <: \{a:Nat\}}}\ \textsc{S-RcdWidth} \qquad \dfrac{}{\mathtt{\{m:Nat\} <: \{\}}}\ \textsc{S-RcdWidth}}{\mathtt{\{x:\{a:Nat,b:Nat\},y:\{m:Nat\}\} <: \{x:\{a:Nat\},y:\{\}\}}}\ \textsc{S-RcdDepth}$$

如果用规则 S-RcdDepth 来改进有单一字段的记录类型(如上面例子所做的，不是改进每一个字段)，我们可用规则 S-Refl 来对其他的字段做一个细微子类型推导：

$$\dfrac{\dfrac{}{\mathtt{\{a:Nat,b:Nat\} <: \{a:Nat\}}}\ \textsc{S-RcdWidth} \qquad \dfrac{}{\mathtt{\{m:Nat\} <: \{m:Nat\}}}\ \textsc{S-Refl}}{\mathtt{\{x:\{a:Nat,b:Nat\},y:\{m:Nat\}\} <: \{x:\{a:Nat\},y:\{m:Nat\}\}}}\ \textsc{S-RcdDepth}$$

还可以用传递规则 S-Trans 将广度子类型化和深度子类型化规则结合起来进一步推导。比如，可以去掉一个类型，同时提升另一个类型，以获得超类型：

$$\dfrac{\dfrac{}{\mathtt{\{x:\{a:Nat,b:Nat\},y:\{m:Nat\}\} <: \{x:\{a:Nat,b:Nat\}\}}}\ \textsc{S-RcdWidth} \qquad \dfrac{\dfrac{}{\mathtt{\{a:Nat,b:Nat\} <: \{a:Nat\}}}\ \textsc{S-RcdWidth}}{\mathtt{\{x:\{a:Nat,b:Nat\}\} <: \{x:\{a:Nat\}\}}}\ \textsc{S-RcdDepth}}{\mathtt{\{x:\{a:Nat,b:Nat\},y:\{m:Nat\}\} <: \{x:\{a:Nat\}\}}}\ \textsc{S-Trans}$$

从最后的记录子类型化规则可看出字段的顺序如何变化对安全使用该记录没什么影响，因为构造好了记录后惟一要做的事情是取出字段，而不是关心字段的顺序。

$$\frac{\{\mathtt{k}_j:\mathtt{S}_j{}^{j\in 1..n}\}\ \text{是}\ \{\mathtt{l}_i:\mathtt{T}_i{}^{i\in 1..n}\}\ \text{的一个置换}}{\{\mathtt{k}_j:\mathtt{S}_j{}^{j\in 1..n}\} <: \{\mathtt{l}_i:\mathtt{T}_i{}^{i\in 1..n}\}} \qquad \text{(S-RcdPerm)}$$

例如：规则 S-RcdPerm 说明{c:Top,b:Bool,a:Nat}是{a:Nat,b:Bool,c:Top}的一个子类型，反之也成立(这说明子类型关系不是反对称的)。

将规则 S-RcdPerm 与 S-RcdWidth 和 S-Trans 结合使用，在记录类型中的任何位置都能减少字段，而不一定非要在末端。

15.2.1 练习[★]：做一推导证明{x:Nat,y:Nat,z:Nat}是{y:Nat}的一个子类型。

S-RcdWidth，S-RcdDepth 和 S-RcdPerm 中每个规则都使记录的使用灵活化了。为便于讨论将它们视为三个独立规则，特别情况下，有的语言只使用了部分规则；比如，Abadi 和 Cardelli 的对象演算(1996)的大部分变化都忽略了广度子类型化规则。然而考虑到实际使用的方便，将它们三者组合为一条宏规则，一次完成 3 件事(该宏规则将在下一章中讨论)。

因为我们使用的是高阶(higher-order)语言，不但数字和记录可作为参数，而且函数也能作为参数传递给其他函数，所以还需要给出关于函数类型的子类型化规则，即必须说明在任何情况下相关的上下文中原本该使用一个不同的函数类型却实际使用了另一种类型的函数也是安全的。

$$\frac{\mathtt{T}_1 <: \mathtt{S}_1 \qquad \mathtt{S}_2 <: \mathtt{T}_2}{\mathtt{S}_1 \to \mathtt{S}_2 <: \mathtt{T}_1 \to \mathtt{T}_2} \qquad \text{(S-Arrow)}$$

注意到对于左端前提的参数类型的子类型关系是相反的(逆变式),而对函数类型,结果类型有着相同的方向(协变式)。规则是这样的,如果我们有一个类型为 $S_1 \to S_2$ 的函数 f,可知 f 接受类型 S_1 的元素,显然,f 也接受任何为 S_1 的子类型 T_1 的元素。f 的类型同时返回了类型 S_2 的元素;这些返回的结果都属于 S_2 的超类型 T_2。那么,类型 $S_1 \to S_2$ 的任何函数 f 也可以被看做是类型 $T_1 \to T_2$ 的函数。

有另一种观点认为,在一个上下文中只要传递给函数的参数全满足 $T_1 <: S_1$ 及返回的结果也全满足 $S_2 <: T_2$,则本该使用类型 $T_1 \to T_2$ 的函数之处使用了类型 $S_1 \to S_2$ 的函数也是安全并允许的。

最后,可找一个所有类型的超类型。这里引入一个新的类型常量 Top,加上一条规则使 Top 为子类型关系的最大元素:

$$S <: Top \qquad \text{(S-Top)}$$

15.4 节中将对 Top 做更深入的讨论。

形式上,子类型关系是给出规则的最小闭关系。图 15.1,图 15.2 和图 15.3 重点讲述了带记录和子类型化的简单类型 λ 演算的完全定义,突出讲述了在这章中添加的语法形式和规则。注意到子类型关系满足自反性和传递性,所以它是一个前序;然而,由于记录置换规则,它并不是偏序:存在许多不同类型的序对,其中一个类型是另一个类型的子类型。

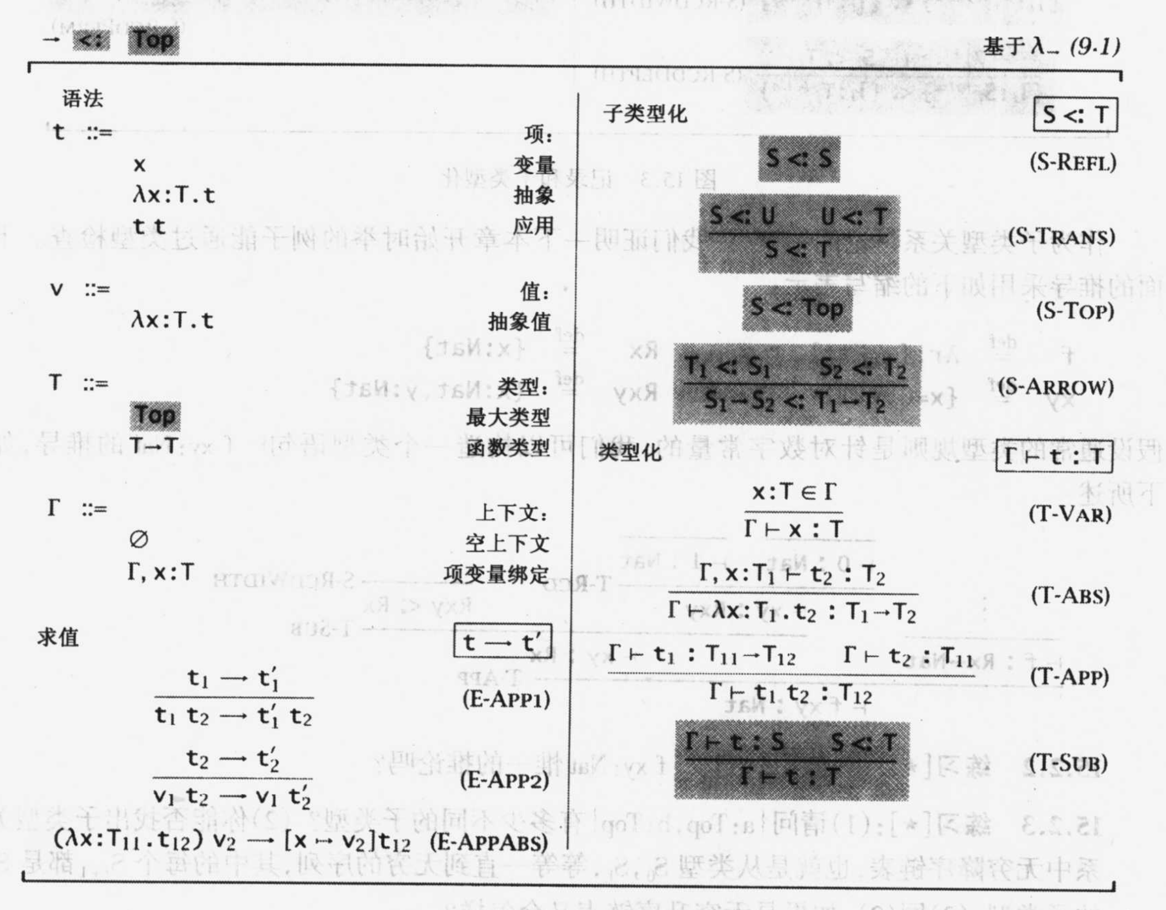


图 15.1 带子类型化的简单类型 λ 演算($\lambda_{<:}$)

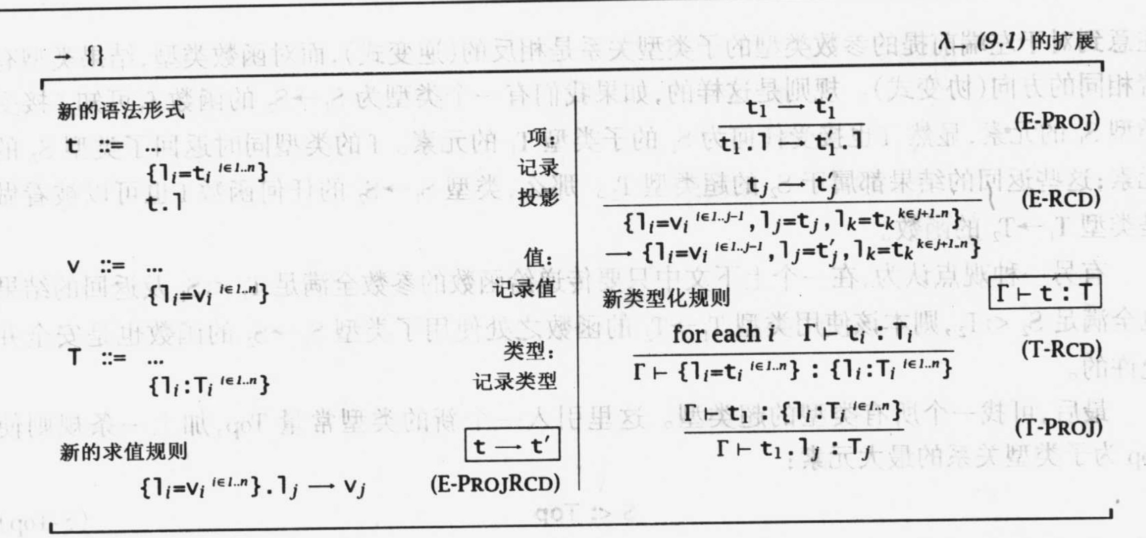


图 15.2 记录规则(同图 11.7)

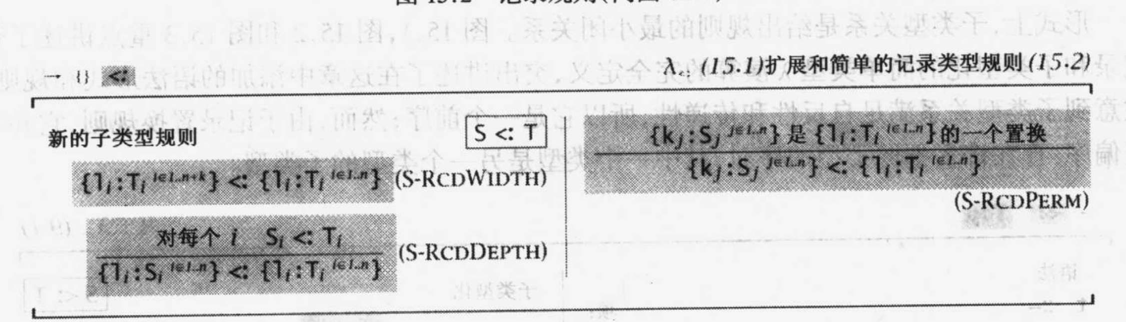


图 15.3 记录和子类型化

作为子类型关系讨论的结束,让我们证明一下本章开始时举的例子能通过类型检查。下面的推导采用如下的缩写表示:

$$
\begin{array}{llllll}
\texttt{f} & \stackrel{\text{def}}{=} & \lambda\texttt{r:\{x:Nat\}. r.x} & \qquad \texttt{Rx} & \stackrel{\text{def}}{=} & \texttt{\{x:Nat\}} \\
\texttt{xy} & \stackrel{\text{def}}{=} & \texttt{\{x=0,y=1\}} & \qquad \texttt{Rxy} & \stackrel{\text{def}}{=} & \texttt{\{x:Nat,y:Nat\}}
\end{array}
$$

假设通常的类型规则是针对数字常量的,我们可以构造一个类型语句⊢ f xy:Nat 的推导,如下所述:

$$
\dfrac{\dfrac{\vdots}{\vdash \texttt{f : Rx}\to\texttt{Nat}} \qquad \dfrac{\dfrac{\dfrac{}{\vdash \texttt{0 : Nat}} \quad \dfrac{}{\vdash \texttt{1 : Nat}}}{\vdash \texttt{xy : Rxy}}\;\text{T-RCD} \qquad \dfrac{}{\texttt{Rxy <: Rx}}\;\text{S-RCDWIDTH}}{\vdash \texttt{xy : Rx}}\;\text{T-SUB}}{\vdash \texttt{f xy : Nat}}\;\text{T-APP}
$$

15.2.2 练习[★]:请问这是语句⊢ f xy:Nat 惟一的推论吗?

15.2.3 练习[★]:(1)请问{a:Top,b:Top}有多少不同的子类型?(2)你能否找出子类型关系中无穷降序链表,也就是从类型 S_0,S_1,等等一直到无穷的序列,其中的每个 S_{i+1} 都是 S_i 的子类型;(3)同(2),如果是无穷升序链表又会怎样?

15.2.4 练习[★]:是否存在一个类型,它是所有类型的子类型?是否存在一个箭头类型(arrow type),它是所有其他箭头类型的超类型?

15.2.5 练习[★★]:假设用11.6节中描述的乘积类型构造子 $T_1 \times T_2$ 来扩展演算,对应于记录的 S-RcdDepth 规则自然地加上一个子类型化规则:

$$\frac{S_1 <: T_1 \qquad S_2 <: T_2}{S_1 \times S_2 <: T_1 \times T_2} \qquad \text{(S-ProdDepth)}$$

如果对乘积加上一个广度子类型化规则是否也是一个好办法?

$$T_1 \times T_2 <: T_1 \qquad \text{(S-ProdWidth)}$$

15.3 子类型化和类型化的性质

确定了子类型化 λ 演算的定义后,现在需要做一些工作以证明它是有意义的,尤其是简单类型 λ 演算的保持和进展定理继续支持了子类型化的存在。

15.3.1 练习[推荐,★★]:在继续阅读之前,让我们先试着预测一下哪里会有困难。特别是,假设我们在定义子类型关系时出现错误,包含了上边提到的规则之外的一个伪规则。系统的哪些属性是错的?另一方面,假设我们遗漏了一条子类型化规则,哪些属性将被破坏?

从记录子类型关系的一个关键属性开始,简单类型 λ 演算中类型关系逆转引理的一个类比[参见引理(9.3.1)]。假如知道某一类型 S 是一箭头类型的子类型,那么子类型化逆转引理告诉我们 S 本身也一定是箭头类型,而且,它还说明箭头的左端是逆变的,右端是协变的。当 S 是记录类型的一个子类型,可采用类似的考虑:S 在某顺序(S-RcdPerm)中具有更多的字段(S-RcdWidth),而且公用字段的类型也存在子类型关系(S-RcdDepth)。

15.3.2 引理[子类型关系的逆转定理]

1. 如果 $S <: T_1 \to T_2$,那么 S 形为 $S_1 \to S_2$,其中 $T_1 <: S_1$ 和 $S_2 <: S_2, T_2$。
2. 如果 $S <: \{l_i : T_i^{\,i \in 1..n}\}$,那么 S 有形为 $\{k_j : S_j^{\,j \in 1..m}\}$,且至少含有标签 $\{l_i^{\,i \in 1..n}\}$,即 $\{l_i^{\,i \in 1..n}\} \subseteq (\{k_j^{\,j \in 1..m}\}$,且对于每个公用标签 $l_i = k_j$,有 $S_j <: T_i$。

证明:留做练习[推荐,★★]。

为了证明在求值中类型被保持了,我们从类型关系的逆转引理开始[参见简单类型 λ 演算定理(9.3.1)]。这里不把引理用最一般的形式描述出来,为满足保持定理的证明需要,给出了一些情况[一般的形式可以从下一章算法化子类型关系中读出,参见定义(16.2.2)]。

15.3.3 引理:

1. 如果 $\Gamma \vdash \lambda x{:}S_1 . s_2 : T_1 \to T_2$,那么 $T_1 <: S_1$ 且 $\Gamma, x{:}S_1 \vdash s_2 : T_2$。
2. 如果 $\Gamma \vdash \{k_a = s_a^{\,a \in 1..m}\} : \{l_i : T_i^{\,i \in 1..n}\}$,那么 $\{l_i^{\,i \in 1..n}\} \subseteq \{k_a^{\,a \in 1..m}\}$ 且对于每个公用标签 $k_a = l_i$,$\Gamma \vdash s_a : T_i$。

证明:对 T-Sub 情况运用引理(15.3.2),对类型化推导进行直接归纳。

接下来,对类型化关系提出代换引理。引理的语句与简单类型 λ 演算[参见引理(9.3.8)]没有变化,证明也几乎完全相同。

15.3.4 引理[代换]:如果对于 $\Gamma, x:S \vdash t:T$,且 $\Gamma \vdash s:S$,那么 $\Gamma \vdash [x \mapsto s]\ t:T$。

证明:对类型化推导进行归纳。还需要 T-Sub 和记录类型化规则 T-Rcd 及 T-Proj 补充新的情况,直接运用归纳假设。余下的证明类似于引理(9.3.8)的证明。

现在,保持定理的语句与前面相同。但它的证明,在某几点上由于子类型化要复杂些。

15.3.5 定理[保持]:如果对于 $\Gamma \vdash t:T$ 且 $t \to t'$,那么 $\Gamma \vdash t':T$。

证明:对类型化推导直接归纳。大多数的情况都与简单类型 λ 演算保持定理的证明相似[参见定理(9.3.9)]。我们需要对记录类型化规则和包含提出新的情况。

情况 T-VAR: $t = x$

不会发生(对变量不存在求值规则)。

情况 T-ABS: $t = \lambda x:T_1.t_2$

不会发生(t 已经是一个值)。

情况 T-APP: $t = t_1\ t_2 \quad \Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11} \quad T = T_{12}$

从图 15.1 和图 15.2 的求值规则可看出 $t \to t'$ 可从三条规则中得出:E-App1,E-App2,和 E-AppAbs。继续分析:

子情况 E-APP1: $t_1 \to t_1' \quad t' = t_1'\ t_2$

结果可从归纳假设和 T-App 中得到。

子情况 E-APP2: $t_1 = v_1 \quad t_2 \to t_2' \quad t' = v_1\ t_2'$

同理。

子情况 E-APPABS: $t_1 = \lambda x:S_{11}.\ t_{12} \quad t_2 = v_2 \quad t' = [x \mapsto v_2]t_{12}$

根据引理[15.3.3(1)],有 $T_{11} <: S_{11}$ 和 $\Gamma, x:S_{11} \vdash t_{12}:T_{12}$。根据 T-Sub,有 $\Gamma \vdash t_2:S_{11}$。据此和代换引理[参见引理(15.3.4)],我们得到 $\Gamma \vdash t':T_{12}$。

情况 T-RCD: $t = \{l_i = t_i\ ^{i \in 1..n}\} \quad \Gamma \vdash t_i : T_i$ 对每个 i 有:
$T = \{l_i : T_i\ ^{i \in 1..n}\}$

左端是一个记录的惟一的求值规则为 E-Rcd。从这个规则的前提,可看出对某个字段 t_j 有 $t_j \to t_j'$。结果可从归纳假设(应用于对应的假设 $\Gamma \vdash t_j:T_j$)和 T-Rcd 中得到。

情况 T-PROJ: $t = t_1.l_j \quad \Gamma \vdash t_1 : \{l_i : T_i\ ^{i \in 1..n}\} \quad T = T_j$

由图 15.1 和图 15.2 的求值规则可知 $t \to t'$ 可由两个规则得出:E-Proj 和 E-ProjRcd。

子情况 E-PROJ: $t_1 \to t_1' \quad t' = t_1'.l_j$

结果由归纳假设和 T-Proj 得出。

子情况 E-PROJRCD: $t_1 = \{k_a = v_a\ ^{a \in 1..m}\} \quad l_j = k_b \quad t' = v_b$

根据引理[15.3.3(2)],我们有$\{l_i^{\,i\in 1..n}\}\subseteq\{k_a^{\,a\in 1..m}\}$和对每个$k_a = l_i$有$\Gamma\vdash v_a : T_i$。特别地,$\Gamma\vdash v_b : T_j$也满足。

情况 T-SUB: $t : S \quad S <: T$

由归纳假设有$\Gamma\vdash t' : S$。由 T-Sub,有$\Gamma\vdash t : T$。

为证明良类型项不会受阻,我们从典型形式的引理开始讨论(如第 9 章),因为它告诉了我们属于箭头和记录类型的值的可能形状。

15.3.6 引理[典型形式]:

1. 如果 v 是类型$T_1\to T_2$的闭值,那么 v 具有形式$\lambda x{:}S_1.t_2$。
2. 如果 v 是类型$\{l_i : T_i^{\,i\in 1..n}\}$的闭值,那么 v 有形式$\{k_j = v_j^{\,a\in 1..m}\}$,其中$\{l_i^{\,i\in 1..n}\}\subseteq\{k_a^{\,a\in 1..m}\}$。

证明:留做练习[推荐,★★★]。

进展定理和它的证明现在已经相当接近我们看到的简单类型λ演算。绝大多数处理子类型化的任务都已经被推向了典型形式的引理,这里仅仅需要一小部分的变化。

15.3.7 定理[进展]:如果t是封闭的、良类型项,那么或者t是一个值,或者存在t′使$t\to t'$。

证明:对类型推导直接进行归纳。变量情况不会出现(因为 t 是封闭的)。因为抽象本身为值,λ 抽象情况很快就能得出结论。剩下的情况更加有趣。

情况 T-APP: $t = t_1\ t_2 \quad \vdash t_1 : T_{11}\to T_{12} \quad \vdash t_2 : T_{11} \quad T = T_{12}$

通过归纳假设,或者t_1是一个值,或者它可以一步求值,如t_2。如果t_1能作为一步,那么规则 E-App1 适用于 t。如果t_1是一个值且t_2可以作为一步,那么规则 E-App2 适用。最后,如果t_1和t_2都是值,那么典型形式引理(15.3.6)告诉我们t_1有形式$\lambda x{:}S_{11}.t_{12}$,规则 E-AppAbs 适用于 t。

情况 T-RCD: $t = \{l_i{=}t_i^{\,i\in 1..n}\} \quad$ 对每个$i\in 1..n$, $\vdash t_i : T_i$
$T = \{l_i{:}T_i^{\,i\in 1..n}\}$

根据归纳假设,每个t_i或是一个值,或可以作为一步求值。如果所有的t_i都是值的话,那么 t 也是一个值。另一方面,如果至少有一个可以作为一步,那么规则 E-Rcd 适用于 t。

情况 T-PROJ: $t = t_1.l_j \quad \vdash t_1 : \{l_i{:}T_i^{\,i\in 1..n}\} \quad T = T_j$

根据归纳假设,t_1或者是一个值,或者可以作为一步求值。如果t_1可以作为一步,那么 t 也可以。如果t_1是一个值,那么用典型形式引理(15.3.6),t_1有形式$\{k_a = v_j^{\,a\in 1..m}\}$且$\{l_i^{\,i\in 1..n}\}\subseteq(\{k_a^{\,a\in 1..m}\}$,且对于每个$l_i = k_j$,有$\vdash v_j : T_i$。特殊情况下,$l_j$属于$t_1$的标签$\{k_a^{\,a\in 1..m}\}$集合,据规则 E-ProjRcd 告诉我们 t 本身也可以作为一步求值。

情况 T-SUB: $\Gamma\vdash t : S \quad S <: T$

结果从归纳假设直接得出。

15.4　Top 类型和 Bottom 类型

最大类型 Top 并不是含子类型化的简单类型 λ 演算必不可少的部分；可以将它去掉而不会破坏系统性质。然而，由于一些原因，大部分文章都包含它的内容。首先，它与大部分面向对象语言中的对象类型相对应。其次，Top 在结合子类型化和参数多态性的更加复杂的系统中是一个方便的技术装置。例如，在 $F_{<:}$ 系统中（参见第 26 章和第 28 章），Top 的出现能够把囿量词还原为通常的非囿量词使系统简化。甚至记录也可以在 $F_{<:}$ 中被编码，进一步使表达简化（至少为了形式学习）；这种编码关键依赖 Top 类型。最后，因为 Top 操作简单易懂，在例子中常起作用，那么就没有什么理由不保留它了。

很自然，我们会问是否也存在含最小元素的子类型关系——类型 Bot，它是所有类型的子类型。回答是：该扩展在图 15.4 中被形式化。

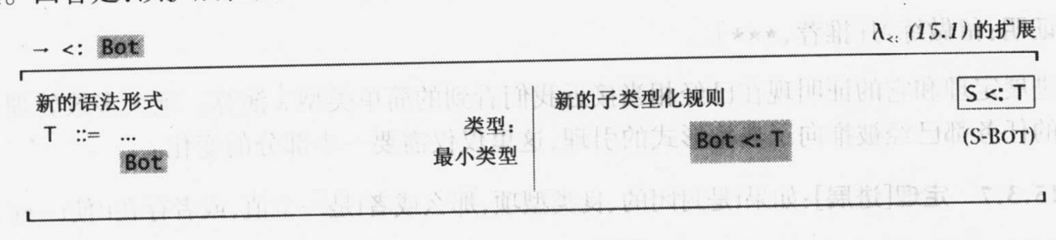


图 15.4　Bottom 类型

我们注意到的第一件事就是 Bot 为空类型——Bot 类型中没有一个闭值。即使有一个的话，假设是 v，那么包含规则加上 S-Bot 就推导出 $\vdash$ v: Top→Top，根据这个推导，典型形式引理［即引理(15.3.6)，在此扩展下依然成立］告诉我们 v 对某个 S_1 和 t_2 必有形式：$\lambda x{:}S_1.t_2$。另一方面，根据包含，还可以得到 $\vdash$ v: {}，通过它，典型形式引理告诉我们 v 必是一个记录。从语法上说，显然 v 不可能既是函数又是记录，因此假设 $\vdash$ v: Bot 是矛盾的。

Bot 的空不是没有用的。相反：Bot 提供了一种非常方便表达某些操作（尤其是产生一个异常或唤醒一次等待）没有返回值的方法。有了这样的表达，类型 Bot 会产生两个好的效果：第一，它提示程序员没有返回结果（若确实返回了一个结果，也是 Bot 型的值）；第二，它发信号给类型检查器，即这种表达式在任何类型的值都适用的上下文中可以安全使用。比如，第 14 章的异常提升项 error 被赋予的类型为 Bot，则下面的项：

```
λx:T.
  if <check that x is reasonable> then
    <compute result>
  else
    error
```

将会是良类型，因为无论正常的结果是什么类型，由包含规则知项 error 总能得到相同的类型，所以如 T-If[①] 要求的那样，if 的两个分支也是相容的。

① 在多态性语言中，如 ML，还可以用 ∀X.X 作为 error 或类似构造子的返回类型。这样是以不同的方式达到了和 Bot 一样的效果：不是给 error 一个类型以使该类型能提升为任意类型，而是给出一个类型选项从而能够实例化为任意类型。尽管它们依据的基础不同，但两种解决方法十分相似：尤其是类型 ∀X.X 也为空时。

可惜的是,Bot 的出现使建造一个系统类型检查器的问题显著复杂化了。一个对含有子类型化的程序语言的简单类型检查算法必须依赖于这样的推论,那就是:“如果一个应用 $t_1 t_2$ 是良类型,那么 t_1 必然有一个箭头类型”。有 Bot 存在时,我们必须将其改为“如果 $t_1 t_2$ 是良类型,那么 t_1 必须是一个箭头类型或者是一个 Bot 类型”,这样的观点会在 16.4 节中详述。在带囿量词的系统中这种复杂性会更加严重。详情参见 28.8 节。

这些复杂的情况说明增加 Bot 类型应比增加 Top 类型更加慎重。在本书以后的部分中,我们将其忽略。

15.5 子类型化及其他特征

当我们考虑扩展带子类型的简单演算使编程语言往成熟方向发展时,每个新特征都要仔细检查它与子类型之间会产生怎样的影响。本节就这一点来考虑一些特征[①]。以后的章节中还会继续讨论,如参数多态性(参见第 26 章和第 28 章)、递归类型(参见第 20 章和第 21 章)及类型算子(参见第 31 章)等方面的特征与子类型之间的更复杂的相互作用。

归属(Ascription)和强制转型(Casting)

在 11.4 节中介绍过归属算子 t as T 作为检查文件的一个形式,允许程序员在程序的文本中记录下一些复杂表达式的子项所具有的特殊类型之类的声明。在本书的例子中,归属也用来控制类型被打印的方式,使类型检查器使用一种更易读的缩写形式来取代它为某项实际计算出的类型。

像 Java 和 C++ 这样具有子类型化的语言,归属变得更有趣。它在这些语言中通常被称为强制转型,并被书写为(T)t。事实上有两种完全不同的形式——称为向上转型和向下转型。前者是直接进行的,而后者,包含动态的类型检测,需要进一步扩展。

在向上转型中项被归为类型的父类型以供类型检查器可以自然地给它设值,所以向上转型是归属算子的实例。我们给出 t 和类型 T,以便“考察”t。类型检查器根据 t 的本来类型,并依据包含规则 T-Sub 和 11.4 节的归属规则来建立如下的推导,证明了 T 确实是 t 的一个类型:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash t : S} \qquad \dfrac{\vdots}{S <: T}}{\Gamma \vdash t : T}\ \text{T-SUB}}{\Gamma \vdash t \text{ as } T : T}\ \text{T-ASCRIBE}$$

其中归属规则为:

$$\frac{\Gamma \vdash t_1 : T}{\Gamma \vdash t_1 \text{ as } T : T} \qquad \text{(T-Ascribe)}$$

向上转型可理解为一种抽象形式:将值的某些部分隐藏起来使之在某些上下文中不受影响。举例来说,如果 t 是一个记录(或更一般化地为一个对象),那么我们可以用向上转型把它的某些字段(或方法)隐藏起来。

---

① 本节讨论的大部分扩展不能在 fullsub 检查器中执行。

另一方面，向下转型能为项赋予类型使类型检查器不能进行静态推导。为能实现向下转型，需对 as 类型化规则做点修改：

$$\frac{\Gamma \vdash t_1 : S}{\Gamma \vdash t_1 \text{ as } T : T} \qquad \text{(T-Downcast)}$$

也就是说，我们检查 t_1 是良类型(即它含有类型 S)，然后给 t_1 赋类型 T，且对 S 与 T 之间的关系不做任何要求。例如，通过向下转型规则可以写出一个带任何参数的函数 f，将其向下转型为一个含数字字段的记录，并将该数字返回：

```
f = λ(x:Top) (x as {a:Nat}).a;
```

实际上，程序员是在告诉类型检查器："我知道(由于某些复杂原因无法根据类型规则来解释)f 总是应用于含数字字段 a 的记录参数；关于这点请相信我"。

当然，盲目的信任将会给语言的安全性带来灾难性的影响：如果程序员在某种程度上犯了一个错误，而且使 f 应用于一个不含字段的记录，结果将可能是完全任意的(视编译器具体情况而定)。换言之，我们的箴言就是："信任，但有依据"。编译时，类型检查器只接受向下转型的类型。然而，在执行的时间里，它会插入一条检查用来验证实际的值是否确为声明的类型。也就是说，对归属的求值规则将不仅仅去掉注释，就像原有归属求值规则所做的：

$$v_1 \text{ as } T \longrightarrow v_1 \qquad \text{(E-Ascribe)}$$

而且会首先把值的实际(运行的)类型与声明类型进行比较：

$$\frac{\vdash v_1 : T}{v_1 \text{ as } T \longrightarrow v_1} \qquad \text{(E-Downcast)}$$

举例来说：给函数 f 赋值{a = 5, b = true}，那么这个规则将(成功地)检查$\vdash$ {a = 5, b = true }：{a: Nat}。另一方面，如果将 f 赋值{b = true}，那么 E-Downcast 规则将不被应用，而且求值规则会在这点上受阻。这种运行时的检查器可以恢复类型保持特性。

15.5.1　练习[★★ ↛]：证明上面的说法。

当然，因为良类型的程序在求值一个坏的向下转型时会受阻，程序无法进行下去。提供向下转型规则的语言通常会采取两种方法来解决：第一，产生一个失败的向下转型以提升一个动态异常供程序捕获和处理(参见第 14 章)；第二，用以下的动态类型检测形式来取代向下转型算子：

$$\frac{\Gamma \vdash t_1 : S \qquad \Gamma, x{:}T \vdash t_2 : U \qquad \Gamma \vdash t_3 : U}{\Gamma \vdash \text{if } t_1 \text{ in } T \text{ then } x \to t_2 \text{ else } t_3 : U} \qquad \text{(T-Typetest)}$$

$$\frac{\vdash v_1 : T}{\text{if } v_1 \text{ in } T \text{ then } x \to t_2 \text{ else } t_3 \longrightarrow [x \mapsto v_1]t_2} \qquad \text{(E-Typetest1)}$$

$$\frac{\nvdash v_1 : T}{\text{if } v_1 \text{ in } T \text{ then } x \to t_2 \text{ else } t_3 \longrightarrow t_3} \qquad \text{(E-Typetest2)}$$

在像 Java 这样的语言中，使用向下转型实际上相当普遍。特别的情况下，向下转型支持一种"贫穷多态性"。例如，像 Set 和 List 这样的"聚集类"，在 Java 中是单一同态的：Java 对每个类型 T 都不提供类型 List T(表示含类型为 T 的元素的列表)，而是仅提供 List，这种类型列表中的元素全是属于最大类型 Object。因为 Object 是 Java 中所有对象(除自身之外)类型的超类

型，所以列表实际上包含了任意类型：当我们想在列表中增加一个元素时，只要用包含规则将其类型提升为 Object。但是，如果要从列表中取出一个元素，类型检查器只知道它的类型为 Object。这种类型不会保证能调用对象的大部分方法，因为类型 Object 只提到了一些基本的方法，如打印及一些所有 Java 对象共享的方法。为了使它真正有用，还是要将其向下转型为需要的类型 T。

一直以来存在一场争论，比如发生在 Pizza（Odersky 和 Wadler，1997），GJ（Bracha，Odersky，Stoutamire 和 Wadler，1998），PolyJ（Myers，Bank 和 Liskov，1997）和 NextGen（Cartwright 和 Steele，1998）之间的，认为最好把 Java 类型系统扩展为实多态性（参见第 23 章），这样比向下转型方法更安全、高效且无需运行检查。然而另一方面，这种扩展由于会影响语言和类型系统的其他许多特性，从而给本已十分庞大的语言增加显著的复杂度（参见 Igarashi，Pierce 和 Wadler，1999，Igarashi，Pierce 和 Wadler，2001）；该事实说明向下转型方法在安全性和复杂性之间提供了一种可靠的折中办法。向下转型在 Java 的反射设置中也起了关键作用。通过反射，程序员能告诉 Java 运行系统动态地装载一个字节码文件并为其中所含的类产生一个实例。显然，类型检查器无法静态地预测到在这一点上加载的类的特征（比如字节码文件可能从网上获得），所以最好的办法就是为新产生的实例赋予最大的类型 Object。还有，为提高效率，必须将新的对象向下转型为期望的类型 T，如果字节码文件提供的类与该类型不匹配，要处理产生的运行异常，然后继续执行转型为 T 后的结果。

在结束向下转型的话题前，还要说明一下有关执行的注释问题。从已给出的规则来说，在语言中加入向下转型机制应当包括为运行系统增加类型检查机制。更糟糕的是，运行时值的表示方式与编译器内部的表示方式不同（尤其是函数被编译成字节码或内部机器指令），这样必须设计一个不同的类型检查器以在动态检查时计算类型。为避免这一点，实际语言给向下的转型带上类型标记——单字符标记（类似于 ML 的数据类型构造子及 11.10 节中介绍的变式标记的方法），这种标记用来捕获编译类型在运行时造成的残余且足以应付动态子类型测试。第 19 章将在这一点上详细分析一个实例。

变式类型

变式的子类型化规则（参见 11.10 节）几乎与记录的子类型化规则相同；惟一的区别就在于，广度规则 S-VariantWidth 中，当子类型向超类型转变时允许增加新的变化类型，而不是减少。直观上说，一个标记表达式 $<l=t>$ 与变式类型 $<l_i:T_i^{i\in 1..n}>$ 之间，如果标签 l 是该类型所列集合 $\{l_i\}$ 的其中一个 l_i，则 $<l=t>$ 是属于该变式类型的；在该集合中增加的标签越多，获得关于元素的信息则会越少。单一变式类型 $<l_1:T_1>$ 能精确说明元素的标签是什么；两个变式类型 $<l_1:T_1, l_2:T_2>$ 说明它表示的元素可能有标签 l_1，也可能有标签 l_2，等等。相反，当使用变式类型的值时，它总是出现在 case 语句中，必须对类型中列出的每个变式对应一个分支，列出的变式越多只会使 case 语句包括一些多余的分支。

把子类型化和变式类型结合起来的另一个结果就是我们可以省去标记结构中的注释部分，就像 11.10 节中一样，直接用 $<l=t>$ 表示 $<l=t>$ as $<l_i:T_i^{i\in 1..n}>$，并且改变标记，用 $<l_1=t_1>$ 进一步精确类型 $<l_1:T_1>$。那么将包含规则加上 S-VariantWidth 规则就可获得任意大的变式类型。

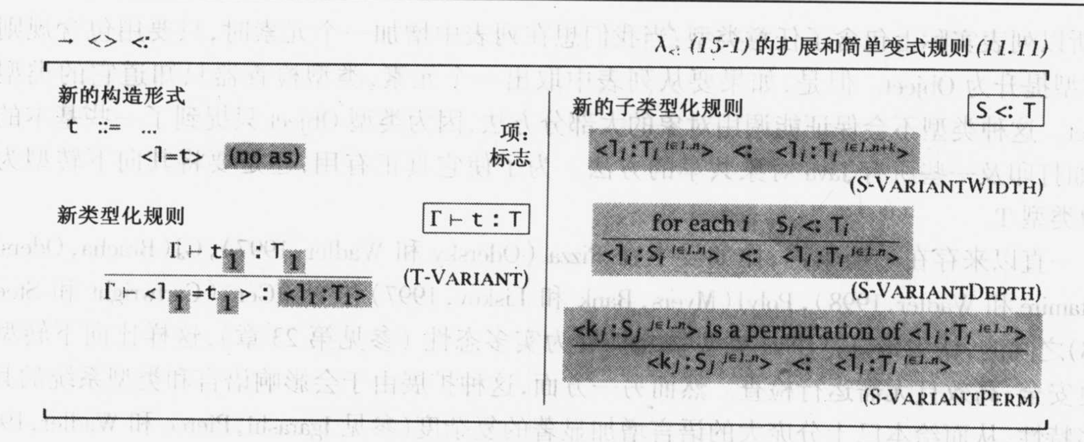


图 15.5 变式类型和子类型

列表类型

我们见到了许多协变式类型构造子(如记录型、变式型及函数型的右端)及一个逆变式类型构造子(箭头型的左端)。List 的构造子也是协变式：如果我们有一个元素类型为 S_1 的列表，且 $S_1 <: T_1$，那么就可以安全地认为列表的元素也是 T_1 类型的：

$$\frac{S_1 <: T_1}{\texttt{List}\ S_1 <: \texttt{List}\ T_1} \quad \text{(S-List)}$$

引用类型(一)

只有部分类型构造子是协变式或逆变式。例如，类型 Ref 的构造子，为了保持类型的安全性必须是不变式：

$$\frac{S_1 <: T_1 \qquad T_1 <: S_1}{\texttt{Ref}\ S_1 <: \texttt{Ref}\ T_1} \quad \text{(S-Ref)}$$

由于 Ref S_1 是 Ref T_1 的一个子类型，我们要求 S_1 和 T_1 在子类型关系中是等价的：任何一个都是另一个的子类型。在 Ref 构造子中记录的字段顺序可以改变，如 Ref{a:Bool,b:Nat} <: Ref{b:Nat,a:Bool}(仅限于此)。

这种子类型化规则要求特别严格，因为类型 Ref T_1 的值在给定上下文中有两种处理方式：读(!)和写(:=)。当它被用做读时，上下文希望能够获得类型为 T_1 的值，所以，如果引用实际上导出了类型 S_1 的值，那么就需要利用 $S_1 <: T_1$ 来避免违背上下文的意思。而另一方面，如果同样的引用单元被用做写，那么上下文提供的新值将是 T_1 型的。如果实际的引用是 Ref S_1，那么其他人可能后来会读到这个值，而且把它当做 S_1 型来使用，这时只有 $T_1 <: S_1$ 才是安全的。

15.5.2 练习[★ ↛]：(1)如果去掉了 S-Ref 的第一个前提，请写出一个产生运行错误类型后失败的小程序(即求值受阻)；(2)如果去掉 S-Ref 的第二个前提，写出运行将失败的程序。

数组类型

显然引用的不变子类型化规则思想也同样适用于数组，因为数组的操作包括反引用和赋值两种形式。

$$\frac{S_1 <: T_1 \qquad T_1 <: S_1}{\texttt{Array } S_1 <: \texttt{Array } T_1} \qquad \text{(S-Array)}$$

有趣的是，Java 还允许数组有协变子类型化规则：

$$\frac{S_1 <: T_1}{\texttt{Array } S_1 <: \texttt{Array } T_1} \qquad \text{(S-ArrayJava)}$$

（在 Java 的语法中，$S_1[\,] <: T_1[\,]$）。该特性最初引入时是为了弥补某些基本操作（如拷贝数组的部分值）的类型化过程中缺少参数多态这点缺陷，但是由于它严重地影响了含有数组程序的性能，所以现在普遍认为它是语言设计中的一个缺陷。原因就是不可靠的子类型化规则造成的不便必须以运行时对任何数组的每次赋值进行检查的方式来弥补，这种检查是为了确保写的每个值都符合数组元素的实际类型（的子类型）。

15.5.3 练习[★★★ ↛]：编写一简短的 Java 程序，要求含有不能通过类型检查的数组类型（要提升 ArrayStoreException 异常）。

引用类型（二）

最早 Reynolds 在语言 Forsythe(1988)中对引用进行了更细致的分析，引入了两个新类型构造子 Source 和 Sink。直观地讲，Source T 具有从单元中读出类型 T 的值的能力（但是不允许赋值），而 Sink T 则是具有向单元中写入类型 T 的值的能力。如果同时赋予读和写的能力，那么 Ref T 就是这两种能力的结合体。

反引用和赋值类型化规则（参见图 13.1）在这里都做了修改，这样只需具备合适的能力：

$$\frac{\Gamma \mid \Sigma \vdash t_1 : \texttt{Source } T_{11}}{\Gamma \mid \Sigma \vdash \texttt{!}t_1 : T_{11}} \qquad \text{(T-Deref)}$$

$$\frac{\Gamma \mid \Sigma \vdash t_1 : \texttt{Sink } T_{11} \qquad \Gamma \mid \Sigma \vdash t_2 : T_{11}}{\Gamma \mid \Sigma \vdash t_1 \texttt{:=} t_2 : \texttt{Unit}} \qquad \text{(T-Assign)}$$

现在，如果我们仅仅具有从单元中读出数值的能力，而且如果这些值保证含有类型 S_1，那么只要 S_1 是 T_1 的子类型，就可以安全"降级"，使其具有读出类型 T_1 的值的能力。也就是说，Source 构造子是协变的：

$$\frac{S_1 <: T_1}{\texttt{Source } S_1 <: \texttt{Source } T_1} \qquad \text{(S-Source)}$$

相反地，往指定的单元中写类型 S_1 的值的能力可以降级为写某个更小的类型 T_1 的值的能力：Sink 构造子是逆变的：

$$\frac{T_1 <: S_1}{\texttt{Sink } S_1 <: \texttt{Sink } T_1} \qquad \text{(S-Sink)}$$

最后，通过子类型化规则，Ref T_1 具备了读与写的能力，这样 Ref 能降级为 Source 或 Sink 类型：

$$\texttt{Ref } T_1 <: \texttt{Source } T_1 \qquad \text{(S-RefSource)}$$

$$\texttt{Ref } T_1 <: \texttt{Sink } T_1 \qquad \text{(S-RefSink)}$$

通道类型

相同的思想（及相同的子类型化规则）已经成为了近来出现的并发程序语言 Pict(Pierce 和

Turner, 2000; Pierce 和 Sangiorgi, 1993)中通道类型处理的基础。从类型的观点来看,一个信息通道的行为与一个引用单元完全相似:它可以被用做读和写,而且,因为很难静态地确定哪个读对应哪个写,所以为了确保类型的安全性,惟一简单的办法就是使所有通过通道的值都属于一个相同的类型。现在,如果我们只赋予他人对通道的写能力,那么他们要将该能力给那些承诺写更小类型的值的其他人才是安全的,这种"输出通道"类型构造子是逆变的。类似地,如果赋予了从通道读的能力,则该能力可降级为读取任意更大类型的值的能力,这种"输入通道"类型构造子是协变的。

基类型

对一个成熟的,具有一套丰富基类型的语言,将初始子类型关系引入这些类型中通常非常方便的。举例来讲,在许多语言中,布尔值 true 和 false 实际上是用数字 1 和 0 来代表的。如果愿意,可以通过引入子类型化公理 Bool <: Nat 向程序员说明。现在我们就可以把 if b then 5 else 0 压缩表达为 5 * b。

15.6 子类型化的强制语义

纵览全章,子类型化给人的感觉是"无语义的"。子类型化的出现并没有改变程序求值的方法,但它使项的类型化更加灵活了。这种解释简单而自然,但灵活的同时也带来了某些性能上的牺牲(特别是对数字计算和记录字段的访问),这不能满足高效执行这一要求。在这里我们将提出一个可供选择的强制语义,并且讨论由此引发的新课题。

子集语义上出现的问题

如在 15.5 节中看到,允许不同基类型之间子类型化非常方便。但基类型之间某些"感觉可靠"的包含可能会给性能造成有害的影响。例如,假如引入规则 Int <: Float,让整数不要写明强制转换就可直接用于浮点计算,也就是说,可以这样写,如 4.5 + 6 代替 4.5 + intToFloat (6)。从子集语义的角度说,这条规则意味着整数集合字面上必须是浮点数集合的子集。但在大多数实际机器上,整数与浮点数的具体表示完全不同:整数通常用两个补码来表示,但浮点数被分为尾数、指数和符号数三部分,加上一些特殊的情况如 NaN(非数字)。

为协调子类型化造成子集语义与实际表示不一致的矛盾,可对数字在表示方式上增加一个标志(或包装):整数可在机器数上增加一个标志(可作为一个独立的字头或作为一个实际的数放在该字的高位上),而浮点数可表示为一个机器浮点数加上一个不同的标志。Float 型可指全部带标志的数值(浮点数和整数),而 Int 只能指带标志的整数。

这样的设计是可靠的:实际上它符合许多现代语言实现的表示方法,而且标志位(或字)在垃圾收集中也用得上。说到底,每个对数字的原语操作都应当包含:对参数的标志检查,几条还原原始数据的指令,一条对实际数字操作的指令,以及几条对结果重加标志的指令等。编译器经过优化后可以减少一些系统开销,但即使用上最新最有效的技术,还是会极大地降低系统性能,尤其在需要处理大量数据时,如遇到绘图计算和科学计算。

当记录与子类型化规则(尤其是置换规则)结合时,会产生另一个性能问题。如一个字段投影的简单求值规则:

$$\{\mathtt{l}_i\mathtt{=v}_i\ ^{i\in 1..n}\}.\mathtt{l}_j \longrightarrow \mathtt{v}_j \qquad \text{(E-ProjRcd)}$$

可以读成“在记录的标签中搜寻 l_j，并产生一个相关的值 v_j。”但在实际执行过程中，不想为了找到需要的标签在运行时间里对整个记录进行线性搜索。一个不带子类型化机制(或带子类型化但不含置换规则)的语言可以做得更好：如果标签 l_j 出现在记录的类型中第三个位置，则可以静态地知道该类型任何运行状态的值都以 l_j 为它们的第三个字段，所以运行时不需要注意这些标签(事实上，我们完全可以忽略它们在运行时间上的表示，而仅将记录编译成元组即可)。然后以间接方式通过一个寄存器指向记录的开头，加上 3 个字的常量偏移以获得 l_j 字段的值。但如果含置换规则，就会破坏这一技术，即使知道记录的值是属于某一类型的，且这一类型 l_j 是出现在第三个相对位置上也没什么用，因为 l_j 字段在记录中实际的位置根本就无法预测。若进一步优化改进及采用运行时间相关策略能缓解该问题造成的影响，但总的说来字段投影在运行时间中仍无法避免搜索上的消耗①。

强制语义

我们可采用一个不同的语义来解决上面这两个问题，用运行时间强制方法取代子类型化将其“编译出局”。如在类型检查的同时，将 Int 型转化为 Float 型，在运行时间物理上将该数的表示形式从机器整数转化为机器浮点数。类似地，记录置换子类型化规则实际使用时可以编译成一段字段顺序经过重排了的代码。原始数字操作和字段存取都可以不经过还原或搜索等开销。

直观上说，带子类型化的语言中的强制语义其实可以表示为一个函数，它将项从该语言的形式转化为一个更低级的不含子类型化规则的语言形式。最终，低级语言可能就是实际处理的机器代码。然而，为说得清楚点，这里的讨论仍停留在一个更抽象的层次上。至于选什么源语言来说明，这里考虑选择一个到目前为止大部分章节中都用到的含子类型和记录型的简单类型化 lambda 演算。对低级目标语言，这里选择含记录型和 Unit 型(用来解释 Top 类型的)纯简单类型lambda 演算。

通常，编译过程包含了三个翻译函数：一个作用于类型；一个用来子类型化；一个用来类型化。对类型，翻译过程是用 Unit 取代 Top。将此函数记为⟦—⟧。

$$\begin{aligned}
[\![\mathtt{Top}]\!] &= \mathtt{Unit}\\
[\![\mathtt{T}_1\to\mathtt{T}_2]\!] &= [\![\mathtt{T}_1]\!]\to[\![\mathtt{T}_2]\!]\\
[\![\{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\}]\!] &= \{\mathtt{l}_i\mathtt{:}[\![\mathtt{T}_i]\!]\ ^{i\in 1..n}\}
\end{aligned}$$

举例来说：⟦Top→{a:Top,b:Top}⟧ = Unit→{a:Unit,b:Unit}(另一种翻译函数也写成⟦—⟧；在上下文中我们一般都会说明指的是哪一种)。

为翻译一个项，必须知道在类型检查时什么位置该使用包含规则，因为这些位置将插入运行时间强制转换。为将结论形式化，有一个方便的办法就是把翻译程序当成类型声明的推导函数。类似地，为产生一个将类型 S 的值转化为类型 T 的值的强制转化函数，不仅需要知道 S 是 T 的子类型，还要弄清楚为什么。为达到这个目的，需要从子类型化推导中产生强制语义。

① 这种情况类似于带允许置换的子类型化的语言中对象的字段和方法的处理。正是这个原因，在 Java 中限制类之间子类型化，让新的字段只能添加到类的最后。接口之间子类型化(及类与接口之间子类型化)可以用置换规则(如果不这样做界面几乎没有什么用了)，故手册上会明确警告从一个界面中查找方法比从类中查找方法要慢。

这里为使翻译形式化,先找一个可以表示推导的符号。符号$\mathcal{C}$:: S <:T 表示"$\mathcal{C}$作为一个子类型推导树,结论是 S <:T"。同样,用$\mathcal{D}$:: Γ⊢ t:T 表示"$\mathcal{D}$是一个类型推导,结论是 Γ⊢ t:T"。

作为子类型声明 S <:T 的推导$\mathcal{C}$来说,先考虑一个产生强制转换⟦$\mathcal{C}$⟧的函数。其实它只是一个从类型⟦ S ⟧到类型⟦ T ⟧的函数(在翻译的目标语言中用 $\lambda_{\rightarrow}$ 表示)。定义会依据最终$\mathcal{C}$中的规则提出来。

$$\left[\!\!\left[\frac{}{\mathtt{T <: T}}\ \text{(S-Refl)}\right]\!\!\right] = \lambda\mathtt{x}{:}[\![\mathtt{T}]\!].\ \mathtt{x}$$

$$\left[\!\!\left[\frac{}{\mathtt{S <: Top}}\ \text{(S-Top)}\right]\!\!\right] = \lambda\mathtt{x}{:}[\![\mathtt{S}]\!].\ \mathtt{unit}$$

$$\left[\!\!\left[\frac{\mathcal{C}_1 :: \mathtt{S <: U} \quad \mathcal{C}_2 :: \mathtt{U <: T}}{\mathtt{S <: T}}\ \text{(S-Trans)}\right]\!\!\right] = \lambda\mathtt{x}{:}[\![\mathtt{S}]\!].\ [\![\mathcal{C}_2]\!]([\![\mathcal{C}_1]\!]\ \mathtt{x})$$

$$\left[\!\!\left[\frac{\mathcal{C}_1 :: \mathtt{T_1 <: S_1} \quad \mathcal{C}_2 :: \mathtt{S_2 <: T_2}}{\mathtt{S_1{\rightarrow}S_2 <: T_1{\rightarrow}T_2}}\ \text{(S-Arrow)}\right]\!\!\right] = \lambda\mathtt{f}{:}[\![\mathtt{S_1{\rightarrow}S_2}]\!].\ \lambda\mathtt{x}{:}[\![\mathtt{T_1}]\!].\ [\![\mathcal{C}_2]\!](\mathtt{f}([\![\mathcal{C}_1]\!]\ \mathtt{x}))$$

$$\left[\!\!\left[\frac{}{\{\mathtt{l}_i{:}\mathtt{T}_i\ ^{i\in 1..n+k}\} <: \{\mathtt{l}_i{:}\mathtt{T}_i\ ^{i\in 1..n}\}}\ \text{(S-RcdWidth)}\right]\!\!\right] = \lambda\mathtt{r}{:}\{\mathtt{l}_i{:}[\![\mathtt{T}_i]\!]\ ^{i\in 1..n+k}\}.\ \{\mathtt{l}_i{=}\mathtt{r.l}_i\ ^{i\in 1..n}\}$$

$$\left[\!\!\left[\frac{\text{for each } i \quad \mathcal{C}_i :: \mathtt{S}_i <: \mathtt{T}_i}{\{\mathtt{l}_i{:}\mathtt{S}_i\ ^{i\in 1..n}\} <: \{\mathtt{l}_i{:}\mathtt{T}_i\ ^{i\in 1..n}\}}\ \text{(S-RcdDepth)}\right]\!\!\right] = \lambda\mathtt{r}{:}\{\mathtt{l}_i{:}[\![\mathtt{S}_i]\!]\ ^{i\in 1..n}\}.\ \{\mathtt{l}_i{=}[\![\mathcal{C}_i]\!](\mathtt{r.l}_i)\ ^{i\in 1..n}\}$$

$$\left[\!\!\left[\frac{\{\mathtt{k}_j{:}\mathtt{S}_j\ ^{j\in 1..n}\} \text{ perm. of } \{\mathtt{l}_i{:}\mathtt{T}_i\ ^{i\in 1..n}\}}{\{\mathtt{l}_i{:}\mathtt{S}_i\ ^{i\in 1..n}\} <: \{\mathtt{l}_i{:}\mathtt{T}_i\ ^{i\in 1..n}\}}\ \text{(S-RcdPerm)}\right]\!\!\right] = \lambda\mathtt{r}{:}\{\mathtt{k}_j{:}[\![\mathtt{S}_j]\!]\ ^{j\in 1..n}\}.\ \{\mathtt{l}_i{=}\mathtt{r.l}_i\ ^{i\in 1..n}\}$$

15.6.1 引理:如果$\mathcal{C}$:: S <:T,则⊢⟦$\mathcal{C}$⟧:⟦ S ⟧→⟦ T ⟧。

证明:直接对$\mathcal{C}$进行归纳可得出。

类型推导也可以用同样的方法来翻译。如果$\mathcal{D}$是语句 Γ⊢ t:T 的推导,则它的翻译⟦$\mathcal{D}$⟧即为类型⟦ T ⟧的目标语言项。这种翻译函数先由 Pennsylvania 大学一个研究小组研究(有 Breazu-Tannen, Coquand, Gunter 及 Scedrov 等,1991),所以给它取名为 Penn 翻译。

$$\left[\!\!\left[\frac{\mathtt{x{:}T} \in \Gamma}{\Gamma \vdash \mathtt{x : T}}\ \text{(T-Var)}\right]\!\!\right] = \mathtt{x}$$

$$\left[\!\!\left[\frac{\mathcal{D}_2 :: \Gamma, \mathtt{x{:}T_1} \vdash \mathtt{t_2 : T_2}}{\Gamma \vdash \lambda\mathtt{x{:}T_1 : T_1{\rightarrow}T_2}}\ \text{(T-Abs)}\right]\!\!\right] = \lambda\mathtt{x}{:}[\![\mathtt{T_1}]\!].\ [\![\mathcal{D}_2]\!]$$

$$\left[\!\!\left[\frac{\mathcal{D}_1 :: \Gamma \vdash \mathtt{t_1 : T_{11}{\rightarrow}T_{12}} \quad \mathcal{D}_2 :: \Gamma \vdash \mathtt{t_2 : T_{11}}}{\Gamma \vdash \mathtt{t_1\ t_2 : T_{12}}}\ \text{(T-App)}\right]\!\!\right] = [\![\mathcal{D}_1]\!]\ [\![\mathcal{D}_2]\!]$$

$$\left[\!\!\left[\frac{\text{for each } i \quad \mathcal{D}_i :: \Gamma \vdash \mathtt{t}_i : \mathtt{T}_i}{\Gamma \vdash \{\mathtt{l}_i{=}\mathtt{t}_i\ ^{i\in 1..n}\} : \{\mathtt{l}_i{:}\mathtt{T}_i\ ^{i\in 1..n}\}}\ \text{(T-Rcd)}\right]\!\!\right] = \{\mathtt{l}_i{=}[\![\mathcal{D}_i]\!]\ ^{i\in 1..n}\}$$

$$\left[\!\!\left[\frac{\mathcal{D}_1 :: \Gamma \vdash \mathtt{t_1} : \{\mathtt{l}_i{:}\mathtt{T}_i\ ^{i\in 1..n}\}}{\Gamma \vdash \mathtt{t_1.l}_j : \mathtt{T}_j}\ \text{(T-Proj)}\right]\!\!\right] = [\![\mathcal{D}_1]\!].\mathtt{l}_j$$

$$\left[\!\!\left[\frac{\mathcal{D} :: \Gamma \vdash \mathtt{t : S} \quad \mathcal{C} :: \mathtt{S <: T}}{\Gamma \vdash \mathtt{t : T}}\ \text{(T-Sub)}\right]\!\!\right] = [\![\mathcal{C}]\!]\ [\![\mathcal{D}]\!]$$

15.6.2 定理：如果$\mathcal{D}::\Gamma \vdash t:T$，那么$[\![\Gamma]\!] \vdash [\![\mathcal{D}]\!]:[\![T]\!]$，其中$[\![\Gamma]\!]$是类型翻译逐点扩展到上下文的形式，并有$[\![\varnothing]\!]=\varnothing$且$[\![\Gamma,x:T]\!]=[\![\Gamma]\!],x:[\![T]\!]$。

证明：根据引理(15.6.1)中对 T-Sub 的情况的分析可以直接对$\mathcal{D}$进行归纳。

定义了这些翻译后，就可以将具有子类型化的高级语言的求值规则省去，而是换做对项的求值：对它们进行类型检查(用高级类型化和子类型化规则)，接着把它们的类型推导解释为低级目标语言，然后用该语言的求值关系来获得它们的操作行为。这种策略实际上被用在一些具有子类型语言的高效执行中，比如像 Yale 编译小组研制的实验性 Java 编译器(League，Shao 和 Trifonov，1999；League，Trifonov 和 Shao，2001)。

15.6.3 练习[★★★ ↛]：用带元组(代替记录类型)的简单类型 λ 演算作为目标语言来修改上述的翻译。检查定理(15.6.2)是否还成立。

吻合性

当给含子类型的语言一个强制语义时，要小心避免一个潜在的缺陷。例如，假设需要对目前含基类型 Int，Bool，Float 和 String 的语言进行扩展。以下原语强制语义全部都有用：

```
[[Bool <: Int]]       =   λb:Bool. if b then 1 else 0
[[Int <: String]]     =   intToString
[[Bool <: Float]]     =   λb:Bool. if b then 1.0 else 0.0
[[Float <: String]]   =   floatToString
```

函数 intToString 和 floatToString 是将数字转化为字符串的原语函数。例如，假设 intToString(1) = “1”，而 floatToString(1.0) = “1.000”。

现在，假设要用强制语义来求值项：

```
(λx:String.x) true;
```

根据上面原始类型的公理可知，该项是可类型化的。但是它的可类型化的方式有两种：(1)用包含规则来将 Bool 型提升为 Int 型，然后到 String 型，这样说明 true 是 String→String 型函数的合适参数；(2)将 Bool 型提升为 Float 型，再转为 String 型。但是若将这些推导翻译为$\lambda_{\rightarrow}$，将会得到不同的结果。如果将 true 强制转化为 Int 型，结果为 1，然后经过 intToString 产生字符串“1”。但是如果将 true 强制转化为 Float 型，那么用 floatToString 将其变为一个 String 型(从类型推导结构中可知 true：String 要经过一个先转换为 Float 型的过程)为“1.000”。但是“1”和“1.000”是完全不同的两个字符串：它们甚至连长度都不同。换言之，选择用什么样的方法证明$\vdash (\lambda x:String.x)true:String$会影响被翻译程序的行为！但是这种选择完全是在编译器的内部(程序员仅仅写出项，而不是推导)，所以我们设计的语言使程序员无法控制甚至是无法预测所写程序的行为。

解决这个问题合适的做法是在翻译函数的定义中加入一个附加要求，称为吻合性。

15.6.4 定义：对结论都为$\Gamma \vdash t:T$的推导$\mathcal{D}_1$和$\mathcal{D}_2$，若产生的翻译$[\![\mathcal{D}_1]\!]$和$[\![\mathcal{D}_2]\!]$为目标语言中行为等价的项，则说从一个语言中的类型推导转换为另一个语言的项的翻译$[\![-]\!]$，是吻合的。

特别说明的是，以上给出的翻译(不含有基类型)是吻合的。若考虑到有基类型的情况(含

有上述公理)，为恢复吻合性，需要改变 floatToString 最初的定义，从而使 floatToString(0.0) = "0" 及 floatToString(1.0) = "1"。

证明吻合性，特别是对更为复杂的语言，是件棘手的事。详见 Reynolds(1980)，Breazu-Tannen 等(1991)，Curien 和 Ghelli (1992)，及 Reynolds (1991)。

15.7 交叉类型和联合类型

在类型语言中增加一个交叉算子可以有力地改进子类型关系。交叉类型是由 Coppo，Dezani，Sallé 和 Pottinger(Coppo 和 Dezani-Ciancaglini，1978；Coppo，Dezani-Ciancaglini 和 Sallé，1979；Pottinger，1980)首次提出的。还可以从 Reynolds(1988，1998b)，Hindley(1992)和 Pierce(1992b)的相关著作中找到介绍。

交叉类型 $T_1 \wedge T_2$ 中可表示项既属于 S 又属于 T，即 $T_1 \wedge T_2$ 是 T_1 和 T_2 序理论上的交(最大下界)。该思想可用三个新的子类型化规则表示：

$$T_1 \wedge T_2 <: T_1 \qquad \text{(S-Inter1)}$$

$$T_1 \wedge T_2 <: T_2 \qquad \text{(S-Inter2)}$$

$$\frac{S <: T_1 \qquad S <: T_2}{S <: T_1 \wedge T_2} \qquad \text{(S-Inter3)}$$

还有一个附加的规则允许交叉类型和箭头类型之间进行自然变换：

$$S \to T_1 \wedge S \to T_2 <: S \to (T_1 \wedge T_2) \qquad \text{(S-Inter4)}$$

这个规则告诉我们，如果知道一个项具有函数类型 $S \to T_1$ 和 $S \to T_2$，那么可将 S 传递给它，得到的既是 T_1 类型又是 T_2 类型。

在带子类型化和交叉类型的简单类型 λ 演算的按名调用变量中，可赋值类型的无类型 λ 项的集合其实就是一个规范化项的集合，也就是说，一个项是可类型化的当且仅当它的求值可以终止，这一结论正反映了交叉类型的优势所在，从中立即可以说明带交叉类型的演算中类型重构问题(参见第 22 章)是不可判定的。

交叉类型的重要性还在于它支持有限重载形式。例如，给加算子赋予类型($Nat \to Nat \to Nat) \wedge (Float \to Float \to Float$)，使之既能作用于自然数也能作用于浮点数(如可以在运行时给参数加上标志位以区分是什么类型，提醒系统选择正确的指令)。

遗憾的是，交叉类型也给语言设计者带来了些难题。迄今为止，语言 Forsythe(Reynolds，1988)在其最一般的形式上包括了交叉类型，这是惟一完全的语言。还有一种证明为更易处理的受限形式，称为精炼类型(Freeman 和 Pfenning，1991；Pfenning，1993b；Davies，1997)。

无独有偶，存在一个非常有用的联合类型，$T_1 \vee T_2$。与和类型及变式类型(它们有时也被称为"联合类型"，容易混淆)不同，$T_1 \vee T_2$ 指的是属于 T_1 的值的集合与属于 T_2 的值的集合的一般联合，不需要给每个元素加上标记来识别它的来源。所以，$Nat \vee Nat$ 实际上仅仅是 Nat 的另一个名称。虽然非拆解的联合类型已经长期在程序分析中扮演了重要的角色(Palsberg 和 Pavlopoulou，1998)，但仅在少数程序语言发挥重要作用(比较著名的是 Algol 68；参见 van Wijngaarden 等，1975)；然而，近年来，它们越来越多地被应用在处理像 XML(Buneman 和 Pierce，1998；Hosoya，Vouillon 和 Pierce，2001)这样"半结构化"数据库格式的类型系统环境中。

拆解联合类型和非拆解联合类，表面上最主要的区别就是后者缺乏任何一种 case 结构：如果只知道一个值 v 有类型 $T_1 \vee T_2$，那么惟一可以对 v 进行的安全操作就是对 T_1 和 T_2 都有意义的操作（比如说，如果 T_1 和 T_2 都是记录，只有将 v 投影到它们的公用字段上才有意义）。在 C 语言中，无标记的联合类型违反了类型安全性，因为它忽视了这个限制，它允许对$T_1 \vee T_2$ 的元素的任何操作，只要对 T_1 或者 T_2 有意义就行。

15.8 注释

在程序设计语言中子类型化这一思想可以追溯到 20 世纪 60 年代的 Simula（Birtwistle，Dahl，Myhrhaug和 Nygaard，1979）语言以及和它相关的领域。首次对它形式化的处理要归功于 Reynolds（1980）和 Cardelli（1984）。

处理记录的类型化及（尤其是）子类型化规则比起大部分其他规则在本书中谈得多了点，它们包括前提中（每个字段对应一个）的可变数字或者附加的一些机制，如字段索引的置换。这些规则还有许多其他的写法，但是遇到的问题要么较复杂，要么只能通过引入非约定的方式（比如，用省略号：“$l_1 : T_1 \cdots l_n : T_n$”）来避免这种情况。为解决这些困难，Cardelli 和 Mitchell 提出了自己的演算：记录操作（1991），就是将产生多字段记录的宏操作每次分解为一个基本的空记录值及一个添加单字段的操作。诸如适当位字段更新和记录连接（Harper 和 Pierce，1991）这样的附加操作也可以在这样的设置中考虑到。这些操作的类型化规则变得有点难懂，特别是出现了参数多态性，所以多数的语言设计者宁愿使用原来的记录形式。然而，Cardelli 和 Mitchell 的系统还是在理论上留下了一个重要的里程碑。另一种基于列变量多态性的记录处理方式已由 Wand（1987，1988，1989b），Rémy（1990，1989，1992）和其他人开发出来，并且它成了 OCaml（Rémy 和Vouillon，1998；Vouillon，2000）中面向对象特征的基础。

> 类型理论所要解决的基本问题就是保证程序有意义。而由类型理论引发的基本问题却是有意义的，程序往往不具备其应有的意义。矛盾越深，类型系统演变得越丰富。
>
> ——Mark Manasse

第 16 章 子类型的元理论[①]

前一章对具有子类型的简单类型 λ 演算的定义并不能马上实现。与其他演算不同,此系统的规则不是语法制导的——它们不能以“自底向上读”的方式来产生类型检查算法。主要的问题出在类型关系的包含规则(T-Sub)和子类型关系的传递规则(S-Trans)。

T-Sub 存在问题的原因是结论中的项为一个裸露的元变量 t:

$$\frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T} \tag{T-Sub}$$

所有的其他类型规则都说明了项为某具体的形式(T-Abs 仅仅应用于 λ 抽象,T-Var 仅仅用于变量,等等),而 T-Sub 却可以应用于任何一种项。这就意味着,如果给出 t,我们试图计算它的类型,那么它将可能适用于规则 T-Sub 或者任何其他结论与 t 的形状相匹配的规则。

S-Trans 存在同样原因的问题:它的结论覆盖了所有其他规则的结论:

$$\frac{S <: U \qquad U <: T}{S <: T} \tag{S-Trans}$$

因为 S 和 T 都是裸露的元变量,所以我们可以将 S-Trans 规则作为任意子类型化语句推导的最终规则。这样,子类型化规则的“自底向上”实现方式永远无法决定是应该使用这条规则还是应该使用另一条与这两个类型(它们在子类型关系中的身份正是需要检查的)匹配且结论更具体的规则。

S-Trans 还存在另一个问题。它的两个前提都提到了元变量 U,但在结论中却没出现。如果按照自底向上的方式读规则,那么只能猜测一个类型 U,然后试图说明 S <: U 和 U <: T。因为有无穷多的 U,所以这样的办法几乎无法成功。

S-Refl 规则同样覆盖了其他子类型化规则的结论。它的问题较之 T-Sub 和 S-Trans 相对要好一些:自反规则没有前提,所以如果它与要证明的子类型化语句匹配,可以立即成功。这也是规则不是语法制导的另一个原因。

所有这些问题的解决办法就是用称为算法子类型化和算法类型化关系来代替普通的(或者声明性的)子类型化和类型化关系,这两个新关系的推论规则集合是语法制导的。接着在这个转换上要说明原有子类型化和类型化关系实际上与算法表示是一致的:语句 S <: T 可以从算法子类型化规则中推导出来,(当且仅当)它可以从声明性的规则中推导出来;而一个项根据算法类型化规则可类型化(当且仅当)在声明性的规则下它是可类型化的。

将在 16.1 节中提出算法类型关系,在 16.2 节中提出算法类型化关系。将在 16.3 节中讨论一个特殊的多分支结构,如 if … then … else 类型检查问题,该问题需要一些附加的结构(在子类关系中的最小上界或合类型)。将在 16.4 节中讨论最小类型 Bot。

---

① 本章学习的演算是含子类型化(参见图 15.1)和记录(参见图 15.3)的简单类型 λ 演算。对应的 OCaml 实现是 rcdsub。16.3 节也处理布尔型和条件型(参见图 8.1);本节的 OCaml 实现是 joinsub。16.4 节将该讨论扩展到 Bot 型;对应的实现是 bot。

16.1　算法子类型化

带子类型化语言的任何实现中关键是检查一个类型是否是另一个类型的子类型的算法。这个子类型检查器将被类型检查器调用,例如,当它遇到一个应用 $t_1\ t_2$,其中 t_1 含有类型 T→U,t_2 含有类型 S 时,它的功能是用来判断语句 S <: T 是否可以从图 15.1 和图 15.3 的子类型化规则中推导出来。它要检查(S,T)是否属于另一种关系,记为 ↦ S <: T(读做"S 在算法上是 T 的一个子类型"),该关系定义的方式是使成员关系能根据类型的结构来简单判断且含有与声明性子类型关系相同的类型序对。声明关系和算法关系之间最显著的区别就是算法关系去掉了 S-Trans 和 S-Refl 规则。

在一开始,我们需要稍微重新组织一下声明系统。如在 15.2 节所见,我们需要利用传递性将记录的子类型推导"黏合在一起",包括深度子类型化、广度子类型化和置换子类型化规则的组合。在去掉 S-Trans 规则之前,必须先增加一个将深度、广度和置换子类型化规则结合起来的规则:

$$\frac{\{l_i{}^{i\in 1..n}\} \subseteq \{k_j{}^{j\in 1..m}\} \qquad k_j = l_i \text{ 意指 } S_j <: T_i}{\{k_j{:}S_j{}^{j\in 1..m}\} <: \{l_i{:}T_i{}^{i\in 1..n}\}} \qquad \text{(S-Rcd)}$$

16.1.1　引理: 如果 S <: T 能够从 S-RcdDepth,S-RcdWidth 和 S-RcdPerm(没有 S-Rcd)的子类型化规则中推导出来,那么它也可以用 S-Rcd(没有 S-RcdDepth,S-RcdWidth 和 S-RcdPerm)推导出来,反之亦然。

证明: 直接对推导进行归纳。

引理(16.1.1)表明有了规则 S-Rcd,就可以不考虑规则 S-RcdDepth,S-RcdWidth 和 S-RcdPerm。图 16.1 总结出了结果系统。

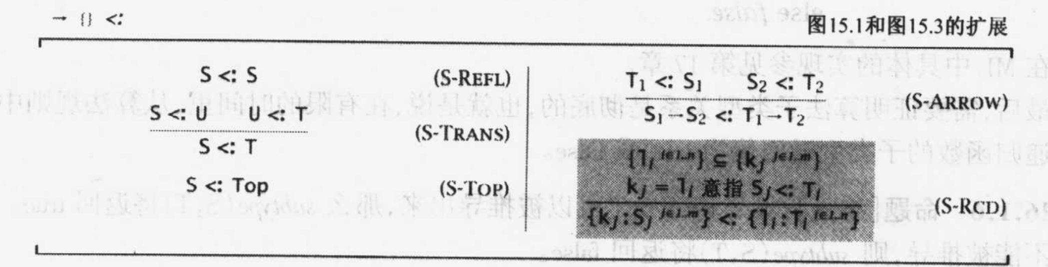


图 16.1　含记录的子类型关系(紧致形式)

接下来我们证明在图 16.1 的系统中,自反性和传递性已经不怎么重要了。

16.1.2　引理:

1. 对每个类型 S,不用 S-Refl 规则就可以推导出 S <: S。
2. 如果 S <: T 是可推导的,那么可以不用 S-Trans 规则就能推导出来。

证明: 作为练习[推荐,★★★]。

16.1.3　练习[★]: 如果添加了类型 Bool,这些特性将如何改变?

这就为我们引出了算法子类型关系的定义。

16.1.4　定义：算法子类型化关系就是图 16.2 中规则封闭的最小类型关系。

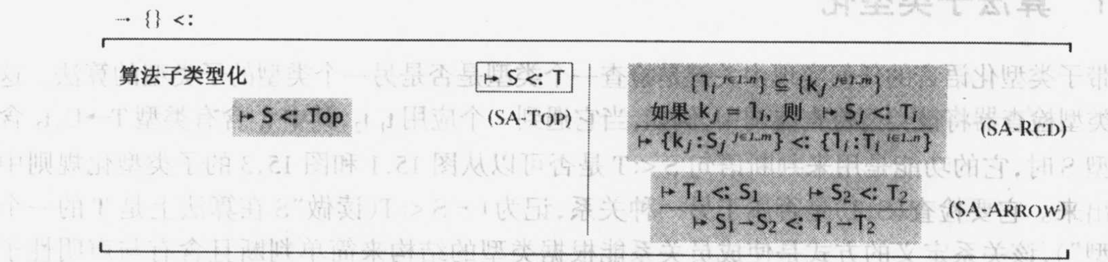


图 16.2　算法子类型化

算法规则是可靠的，因为每个能从算法规则中推导出来的语句也能够从声明性规则中推导出来(算法规则并没有得出什么新的结论)；而且它还是完备的，因为每个能从声明性规则中推导出来的语句也可以从算法规则中推导出来(算法规则能证明以前证明过的所有结论)。

16.1.5　命题[可靠性和完备性]：S <: T 当且仅当 ⊢ S <: T。

证明：用前面两个引理中的一个对推导直接进行归纳。

目前，作为语法制导的算法规则，可以直接用做检查算法子类型关系(并且同样适用于声明性子类型关系)的算法。此算法用更为常规的伪代码表示如下：

```
subtype(S,T) = if T = Top, then true
               else if S = S1→S2 and T = T1→T2
                 then subtype(T1,S1) ∧ subtype(S2,T2)
               else if S = {kj:Sj j∈1..m} and T = {li:Ti i∈1..n}
                 then  {li i∈1..n} ⊆ {kj j∈1..m}
                       ∧ for all i there is some j ∈ 1..m with kj = li
                         and subtype(Sj,Ti)
               else false.
```

算法在 ML 中具体的实现参见第 17 章。

最后，需要证明算法子类型关系是彻底的，也就是说，在有限的时间里，从算法规则中推导出的递归函数的子类型返回值为 true 或 false。

16.1.6　命题[终止]：如果 ⊢ S <: T 可以被推导出来，那么 *subtype*(S,T)将返回 true。如果不能被推导，则 *subtype*(S,T)将返回 false。

该命题加上算法规则的可靠性和完备性，本质上说明子类型函数是对声明性子类型关系的一个决策过程。

证明：第一条命题容易证明(直接对 ⊢ S <: T 的推导进行归纳)。反之，如果 *subtype*(S,T)的返回值是 true，也容易证明 ⊢ S <: T。因此，为了证明第二条，必须证明 *subtype*(S,T)总是有返回值，也就是说：它不能发散。为此，可以做的就是让输入序对 S 和 T 的长度总和是严格地大于算法进行的任意递归调用的参数长度总和。因为这个总和总是正的，所以递归调用的无穷序列是不可能出现的。

读者可能会产生疑问：本节为何不省去一些工作，直接将子类型关系的算法定义作为正式的定义提出，甚至不需要提到声明性子类型关系？回答是“肯定”的。如果愿意，定义演算时当

然可以直接将算法定义作为正式的定义。然而,实际上这样做并不会省去太多的工作,因为,为说明类型化关系(依赖子类型关系的)的行为正确,需要知道子类型化为自反的和传递的,而要证明这些需要的工作量或多或少地与这里所做的相同(另一方面,语言的定义通常都采用一种类型化关系的算法表示。第 19 章将对此举一个例子)。

16.2 算法类型化

解决了子类型关系,现在需要对类型关系进行同样的处理。如本章开始所述,惟一的非语法制导类型化规则就是 T-Sub,所以这是必须进行处理的。如前几节对 S-Trans 所作的工作,我们不能简单地删除包含规则:首先必须检查该规则在类型化的哪个位置起到了关键作用;然后要找到其他的类型化规则,按照语法制导的方法达到与包含规则相同的效果。

显然,包含规则的一个重要作用就是解决函数需要的类型与参数实际类型之间的差异。一个项:

```
(λr:{x:Nat}. r.x) {x=0,y=1}
```

没有包含规则就无法类型化。

也许让人吃惊的是,这可能是包含规则在类型化中惟一起重要作用的地方。在所有其他类型化证明用到了包含规则的情况中,都可以通过另一种推导来证明包含规则能得出的结论,在另一种推导中,包含被移到了树的根部“搁置”起来。要知道为什么,可以做一个尝试:将一些含有包含规则的类型化推导轮流使用别的类型化规则进行,并想一想若是别的类型规则推导的上一层子推导是以 T-Sub 结束的,那么是否可以重新组织一下该规则的推导方式,使其上层子推导不再以 T-Sub 结束。

例如,假设我们给出一个以 T-Abs 结束的类型推导,它的直接上一层子推导以 T-Sub 结束:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, x{:}S_1 \vdash s_2 : S_2} \qquad \dfrac{\vdots}{S_2 <: T_2}}{\Gamma, x{:}S_1 \vdash s_2 : T_2}\ (\text{T-Sub})}{\Gamma \vdash \lambda x{:}S_1.s_2 : S_1 \to T_2}\ (\text{T-Abs})$$

这样的推导可以重新整理,以使包含规则放在抽象规则的后面且得到相同的结论:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, x{:}S_1 \vdash s_2 : S_2}}{\Gamma \vdash \lambda x{:}S_1.s_2 : S_1 \to S_2}\ (\text{T-Abs}) \qquad \dfrac{\dfrac{}{S_1 <: S_1}\ (\text{S-Refl}) \qquad \dfrac{\vdots}{S_2 <: T_2}}{S_1 \to S_2 <: S_1 \to T_2}\ (\text{S-Arrow})}{\Gamma \vdash \lambda x{:}S_1.s_2 : S_1 \to T_2}\ (\text{T-Sub})$$

更有意思的情况是规则 T-App。这里有两个子推导,每个都可能以 T-Sub 结束。先来看一下包含规则出现在左边子推导的最后情况:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash s_1 : S_{11} \to S_{12}} \qquad \dfrac{\vdots}{S_{11} \to S_{12} <: T_{11} \to T_{12}}\ (\text{S-Arrow})}{\Gamma \vdash s_1 : T_{11} \to T_{12}}\ (\text{T-Sub}) \qquad \dfrac{\vdots}{\Gamma \vdash s_2 : T_{11}}}{\Gamma \vdash s_1\ s_2 : T_{12}}\ (\text{T-App})$$

根据前一节的结论，我们可以设想得出 $S_{11}\to S_{12} <: T_{11}\to T_{12}$ 的最后规则既不是 S-Refl 也不是 S-Trans。根据结论的形式，这个规则只能是 S-Arrow。

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma\vdash s_1 : S_{11}\to S_{12}} \qquad \dfrac{\dfrac{\vdots}{T_{11} <: S_{11}} \quad \dfrac{\vdots}{S_{12} <: T_{12}}}{S_{11}\to S_{12} <: T_{11}\to T_{12}}\,\text{(S-Arrow)}}{\Gamma\vdash s_1 : T_{11}\to T_{12}}\,\text{(T-Sub)} \qquad \dfrac{\vdots}{\Gamma\vdash s_2 : T_{11}}}{\Gamma\vdash s_1\ s_2 : T_{12}}\,\text{(T-App)}$$

在消去的 T-Sub 实例的改写中产生了一个有意思的结果：

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma\vdash s_1 : S_{11}\to S_{12}} \qquad \dfrac{\dfrac{\vdots}{\Gamma\vdash s_2 : T_{11}} \quad \dfrac{\vdots}{T_{11} <: S_{11}}}{\Gamma\vdash s_2 : S_{11}}\,\text{(T-Sub)}}{\Gamma\vdash s_1\ s_2 : S_{12}}\,\text{(T-App)} \qquad \dfrac{\vdots}{S_{12} <: T_{12}}}{\Gamma\vdash s_1\ s_2 : T_{12}}\,\text{(T-Sub)}$$

S-Arrow 原来实例右边子推导被往下推到树的底部，而此处 T-Sub 的新实例提升了整个应用节点的类型。另一方面，左边的子推导被向上推到了参数 s_2 的推导里。

假设，我们想重置的 T-Sub 实例发生在 T-App 实例右边的子推导中：

$$\dfrac{\dfrac{\vdots}{\Gamma\vdash s_1 : T_{11}\to T_{12}} \qquad \dfrac{\dfrac{\vdots}{\Gamma\vdash s_2 : T_2} \quad \dfrac{\vdots}{T_2 <: T_{11}}}{\Gamma\vdash s_2 : T_{11}}\,\text{(T-Sub)}}{\Gamma\vdash s_1\ s_2 : T_{12}}\,\text{(T-App)}$$

对 T-Sub 实例惟一能做的就是将其移到左边子推导中，部分颠倒前面的变换：

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma\vdash s_1 : T_{11}\to T_{12}} \qquad \dfrac{\dfrac{\vdots}{T_2 <: T_{11}} \quad \dfrac{}{T_{12} <: T_{12}}\,\text{(S-Refl)}}{T_{11}\to T_{12} <: T_2\to T_{12}}\,\text{(S-Arrow)}}{\Gamma\vdash s_1 : T_2\to T_{12}}\,\text{(T-Sub)} \qquad \dfrac{\vdots}{\Gamma\vdash s_2 : T_2}}{\Gamma\vdash s_1\ s_2 : T_{12}}\,\text{(T-App)}$$

所以，可以看出将一个应用的结果类型提升的包含规则可越过 T-App 规则向下移，但将参数类型与函数定义域类型进行匹配的包含规则不能被删除。它可以从一个前提移至另一个前提[通过提升参数类型使之与函数的定义域匹配，或者提升函数的类型(通过降低参数类型)，这样函数希望获得的参数类型正是我们实际赋予的]，但是不能将包含规则全部去掉。之所以这样说，是因为包含规则解决类型之间差异的作用对整个系统的性能都是十分重要的。

还需要考虑的情况，是推导的最终规则及该推导的直接上一层子推导使用的都是包含规则的地方。如果有这种情况，相邻使用的包含规则可以结合为一个，即以下形式的推导：

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash s : S} \quad \dfrac{\vdots}{S <: U}}{\Gamma \vdash s : U}\ (\text{T-Sub}) \qquad \dfrac{\vdots}{U <: T}}{\Gamma \vdash s : T}\ (\text{T-Sub})$$

可以改写为：

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash s : S} \qquad \dfrac{\dfrac{\vdots}{S <: U} \quad \dfrac{\vdots}{U <: T}}{S <: T}\ (\text{S-Trans})}{\Gamma \vdash s : T}\ (\text{T-Sub})$$

16.2.1 练习[★ ↛]：为了得出尝试结果，想一想在推导中若将 T-Sub 用在 T-Rcd 或 T-Proj 之前，该对推导做怎样类似的调整？

重复地运用这些变换，可以将任意类型化推导改写为一种 T-Sub 只出现在两个位置的特殊形式：一处在应用的子推导右边最后位置，另一处在整个推导的最后。而且，如果只是简单地把处在最后的规则删除，不会产生非常有害的结果：仍然会有一个推导给同一项赋予类型，惟一不同的是所赋的类型可能更小（也就是说是更好的！）。还剩一个位置——出现应用的地方仍可以使用包含规则。为处理这种情况，我们可以将应用规则用一个稍微更有力规则代替：

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_2 \qquad T_2 <: T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}}$$

它将包含规则的一条实例作为前提包括进来。所有由包含领先应用的子推导都可用该规则来代替，这样可以完全不用 T-Sub 规则了。而且，修改的应用规则是语法制导的，所以结论中项的形状不会与其他的规则重叠。

这种变换产生了语法制导类型规则的集合，它可以同以前的类型规则一样给同一个项赋予类型。这些规则可用下述定义概括。如对算法子类型化规则所做的一样，把算法关系用一个有趣的符号“$\mapsto$”表示，即 $\Gamma \mapsto t:T$，以区别声明性关系。

16.2.2 定义：算法类型关系是图 16.3 中规则的最小闭关系。应用规则的前提 $T_1 = T_{11} \to T_{12}$ 是进行类型检查过程中的一个简单而明确的关于操作顺序的提示：首先计算 t_1 的类型 T_1，然后检查 T_1 是否有形式 $T_{11} \to T_{12}$，等等。如果将第一个前提用 $\Gamma \mapsto t_1 : T_{11} \to T_{12}$ 替代，规则将会有相同的作用。对 TA-Proj 也是如此。应用规则中子类型化的前提也可以用一个有趣的符号“$\mapsto$”来表示；既然子类型化的算法表示和声明性表示是等价的，那么究竟选哪一个视不同口味而定。

16.2.3 练习[★ ↛]：请说明在求值中若发现有分别带算法类型 S 和 T 的两个项 s 和 t 满足 $s \to^* t$ 及 $T <: S$，但 $S \not<: T$ 这样的条件，则由算法规则赋给项的类型会变小。

现在需要从形式上检查算法类型规则与原有的声明性规则之间是否符合（上面列举的关于类型推导的变换不太正规，不能作为证明。可以将它们归并为一个，但是那将变得没必要的冗长和繁琐：和往常一样对推导进行归纳更简单些）。和子类型化关系一样，可以断定算法类型关系与原有的声明性规则一样可靠且完备。

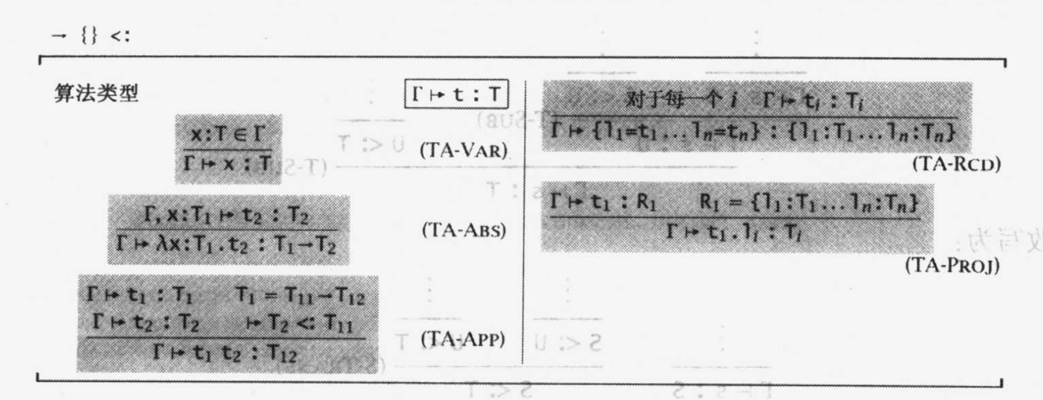


图 16.3 算法类型化

可靠性并没有改变:所有由算法规则推导出的类型语句也可以由声明性规则推导出来。

16.2.4 定理[可靠性]:如果 $\Gamma \mapsto t : T$,那么 $\Gamma \vdash t : T$。

证明:直接对算法类型化推导进行归纳。

完备性的证明看起来会有些不同。因为一般的类型关系能给项赋予很多类型,而算法类型关系却最多赋予一个类型(这样容易检查)。所以定理(16.2.4)的直接逆命题显然是不成立的。可以证明,如果 t 在一般的类型化规则之下有类型 T,那么它在算法规则之下就可以有一个更佳的类型 S,而且 S <: T。换言之,算法规则赋给每个可类型化的项最小类型。完备性定理通常也被称为最小类型定理,因为它[当与定理(16.2.4)结合时]可以说明每个可类型化的项在声明性系统中都有一个最小类型。

16.2.5 定理[完备性,或最小类型]:如果 $\Gamma \vdash t : T$,对 $S <: T$ 有 $\Gamma \mapsto t : S$。

证明:留做练习[推荐,★★]。

16.2.6 练习[★★]:如果去掉了箭头子类型化规则 S-Arrow,保留了剩下相同的声明性子类型化和类型化规则,那么系统还有最小类型化性质吗?如果有,请证明之;如果没有,请举一个无最小类型的可类型化项的例子。

16.3 合类型和交类型

在一个具有子类型的语言中,多结果分支的类型检查表达式,如条件表达式或 case 表达式,需要一些附加机制。例如,回想一下条件表达式的声明性类型化规则:

$$\frac{\Gamma \vdash t_1 : \texttt{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : T} \quad \text{(T-If)}$$

它要求两个分支的类型完全相同,并且要把类型赋给整个条件表达式。但是,由于有包含规则,存在很多种给两个分支赋予同样类型的方法。比如:

```
if true then {x=true,y=false} else {x=false,z=true};
```

的类型为{x:Bool},因为 then 的分支有最小类型{x:Bool,y:Bool},只要用 T-Sub 规则就可以将其提升为{x:Bool},同理,else 分支含有最小类型{x:Bool,z:Bool},可能被提升为{x:Bool}。它

同样还含有类型{x:Top}和{ }。实际上,只要是{x:Bool,y:Bool}和{x:Bool,z:Bool}的超类型都行。所以整个条件表达式的最小类型当然就是{x:Bool,y:Bool}和{x:Bool,z:Bool}超类型中最小的那个类型,即{x:Bool}。通常,为计算任意条件表达式的最小类型,需要分别计算 then 和 else 分支的最小类型,然后再计算这些最小类型的最小公用超类型。这个类型通常被称为分支类型的合,因为它对应于偏序的两个元素的合。

16.3.1 定义:对类型 J 和类型对 S 和 T,如果有 S <: J,T <: J,且对所有的类型 U,如果 S <: U且 T <: U,那么有 J <: U,则称 J 为 S 和 T 的合类型,记为 S ∨ T = J。同理,对类型 M,如果 M <: S,M <: T,且对所有的类型 L,如果 L <: S 且 L <: T,那么 L <: M 则 M 称为 S 和 T 的交类型,记为 S ∧ T = M。

由于受含子类型化的特殊语言对子类型关系的定义方式的限制,每一对类型不一定能找到合类型。对一个子类型关系系统来说,如果所有的 S 和 T,都存在类型 J 为 S 和 T 的合类型,则系统存在合类型。同样,如果所有的 S 和 T,都存在类型 M 为 S 和 T 的交类型,则该系统存在交类型。

在本节[①] 中,所考虑的子类型关系有合类型,没有交类型。例如,类型{}和 Top→Top 不含有任何公用子类型,所以它们当然没有最大的子类型。但是,却含有稍弱的性质。一对类型 S 和 T,如果有某个类型 L,满足 L <: S 且 L <: T,那么 S 和 T 称为有下界的。对一个子类型关系系统,如果所有类型 S 和 T 都有下界,即有类型 M 为 S 和 T 的交类型,则该系统存在有界交。

合类型和交类型不一定是惟一的。比如{x:Top,y:Top}和{y:Top,x:Top}都是{x:Top,y:Top,z:Top}和{x:Top,y:Top,w:Top}的合类型。然而,同一对类型的两个不同合类型(或交类型)必须互为子类型。

16.3.2 命题 [合类型和有界交类型的存在性]:

1. 每一对类型 S 和 T,存在类型 J 满足 S ∨ T = J。
2. 每一对有公用子类型的类型 S 和 T,存在类型 M 满足 S ∧ T = M。

证明:留做练习[推荐,★★★]。

有了合操作,可以在子类型关系中出现 if 结构的地方给出一个算法规则:

$$\frac{\Gamma \vdash t_1 : T_1 \qquad T_1 = \mathtt{Bool} \qquad \Gamma \vdash t_2 : T_2 \qquad \Gamma \vdash t_3 : T_3 \qquad T_2 \vee T_3 = T}{\Gamma \vdash \mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3 : T} \quad \text{(TA-If)}$$

16.3.3 练习[★★]:if true then false else {}的最小类型是什么?它是我们想要的答案吗?

16.3.4 练习[★★★★]:若将合类型和交类型的算法推广到带引用类型强制性语言中(如 15.5 节所述),容易办到吗?若是对 15.5 节的引用类型(此处用协变的 Source 和逆变的 Sink 类型对不变的 Ref 进行了改进)进行处理会如何?

---

① 就是图 15.1 和图 15.3 中定义的用 Bool 扩展的关系。Bool 型的子类型化过程很简单:因为在说明子类型关系中没有加入其他的规则,所以它有惟一的超类型为 Top。

16.4 算法类型化和 Bottom 类型

子类型关系中如果添加了最小类型 Bot(参见 15.4 节),子类型化算法和类型化算法必须进行一点扩展。将一条规则(明显的一条)加入算法子类型关系中:

$$\mapsto \texttt{Bot} <: \texttt{T} \qquad \text{(SA-Bot)}$$

以及将两个复杂些的规则加入到算法类型化关系中:

$$\frac{\Gamma \mapsto \texttt{t}_1 : \texttt{T}_1 \qquad \texttt{T}_1 = \texttt{Bot} \qquad \Gamma \mapsto \texttt{t}_2 : \texttt{T}_2}{\Gamma \mapsto \texttt{t}_1\ \texttt{t}_2 : \texttt{Bot}} \qquad \text{(TA-AppBot)}$$

$$\frac{\Gamma \mapsto \texttt{t}_1 : \texttt{R}_1 \qquad \texttt{R}_1 = \texttt{Bot}}{\Gamma \mapsto \texttt{t}_1.\texttt{l}_i : \texttt{Bot}} \qquad \text{(TA-ProjBot)}$$

子类型化规则的含义是清晰的。类型化规则背后的含义是:在声明性系统中,类型 Bot 的元素可作为任意类型的参数(用包含规则将 Bot 类型提升为想要的任何函数类型),并假设其结果是任意其他的类型(对投影也类似)。

16.4.1 练习[★]:假设在语言中含有条件表达式,是否还需要为 if 添加其他算法类型规则呢?

本节内容中用来支持 Bot 的附加规则不太复杂。在 28.8 节中,还会看到 Bot 与囿量词组合后出现的极为复杂的情况。

第 17 章　子类型化的 ML 语言实现

本章将用一个额外的支持子类型化机制，尤其是用一个检查子类型关系的函数，对第 10 章中简单类型化演算的 OCaml 实现进行扩展。

17.1　语法

类型和项的数据类型定义遵循图 15.1 和图 15.3 中的抽象语法。

```
type ty =
    TyRecord of (string * ty) list
  | TyTop
  | TyArr of ty * ty

type term =
    TmRecord of info * (string * term) list
  | TmProj of info * term * string
  | TmVar of info * int * int
  | TmAbs of info * string * ty * term
  | TmApp of info * term * term
```

对照纯简单类型 λ 演算，新的构造子是类型 TyTop，类型构造子 TyRecord，还有项构造子 TmRecord 和 TmProj。我们用最简单的方法表示记录及其类型，如作为字段名和相关的项或类型的列表。

17.2　子类型化

在 16.1 节提到的算法子类型关系的伪代码表示能够翻译为 OCaml，如下所示：

```
let rec subtype tyS tyT =
   (=) tyS tyT ||
   match (tyS,tyT) with
     (TyRecord(fS), TyRecord(fT)) →
       List.for_all
         (fun (li,tyTi) →
            try let tySi = List.assoc li fS in
                subtype tySi tyTi
            with Not_found → false)
         fT
   | (_,TyTop) →
       true
   | (TyArr(tyS1,tyS2),TyArr(tyT1,tyT2)) →
       (subtype tyT1 tyS1) && (subtype tyS2 tyT2)
   | (_,_) →
       false
```

我们对伪代码表示中的算法做了一些改变,在开始的地方增加了自反性检查[(=)操作符代表一般的等式;它在这里以前缀的形式出现,因为在一些其他的子类型实现中,它会被不同的比较函数所代替。||操作符是一种简化的布尔形式或表示:如果第一个分支是 true,那么第二个分支将不被考虑]。严格地说来,这种检查不是必需的,但它确是实际编译程序里重要的优化手段。在大多数实际程序中,子类型很少使用,也就是说,多数情况下子类型检查被调用时,比较的两个类型实际上是相同的。并且,如果类型的表示方式使得在结构上同构的类型能够保证具有物理上相同的表式,例如在构造类型时使用 hash consing (Goto, 1974; Appel 和 Gonçalves,1993),那么这种检查只是一个指令。

记录的子类型规则自然地涉及到一定数量繁杂的列表。List.for_all 将谓词(它的第一个参数)应用于列表中的每一个成员,当且仅当所有这些应用返回值为 true,它的返回值才为 true。List.assoc li fS,是在列表字段 fS 中查找标签 li 并且返回相关的字段类型 tySi;如果标签 li 不在 fS 中,就提升 Not_found 异常,以让我们捕获并转为 false 响应。

17.3 类型化

类型检查函数是对更早执行的 typeof 函数的直接扩展。主要修改的是应用子句,在参数类型和函数期望的类型之间增加了子类型检查。并且我们也为记录构造和投影增加了两个新的子句:

```
let rec typeof ctx t =
  match t with
    TmRecord(fi, fields) →
      let fieldtys =
        List.map (fun (li,ti) → (li, typeof ctx ti)) fields in
      TyRecord(fieldtys)
  | TmProj(fi, t1, l) →
      (match (typeof ctx t1) with
          TyRecord(fieldtys) →
            (try List.assoc l fieldtys
             with Not_found → error fi ("label "^l^" not found"))
        | _ → error fi "Expected record type")
  | TmVar(fi,i,_) → getTypeFromContext fi ctx i
  | TmAbs(fi,x,tyT1,t2) →
      let ctx' = addbinding ctx x (VarBind(tyT1)) in
      let tyT2 = typeof ctx' t2 in
      TyArr(tyT1, tyT2)
  | TmApp(fi,t1,t2) →
      let tyT1 = typeof ctx t1 in
      let tyT2 = typeof ctx t2 in
      (match tyT1 with
          TyArr(tyT11,tyT12) →
            if subtype tyT2 tyT11 then tyT12
            else error fi "parameter type mismatch"
        | _ → error fi "arrow type expected")
```

记录子句引入了以前没见过的一些 OCaml 的特征。TmRecord 子句是从名和项 fields 的列表中计算出字段名和类型 fieldtys,主要通过 List.map 将函数:

```
fun (li,ti) → (li, typeof ctx ti)
```

依次应用每个名字和项上。在 TmProj 子句中，再次使用 List.assoc 来查找所选字段的类型；如果它提升了 Not _ found 异常，将给出自定义的出错信息(^ 代列表一个字符串)。

17.3.1　练习[★★★]:16.3 节说明了将布尔类型和条件表达式加到带子类型化的语言中，是如何需要额外的支持函数来计算给定类型序对的最小上界。命题(16.3.2)的证明给了必要算法的数学描述。

joinexercise 类型检查器是带子类型、记录和条件表达式简单类型 λ 演算的不完全实现：提供了基本的解析和打印功能，但是 TmIf 子句不在 typeof 函数中出现，它所依赖的 join 函数也不出现。将布尔型和条件表达式(及合类型和交类型)加到该实现过程中。

17.3.2　练习[★★]:按照 16.4 节中的描述，将最小 Bot 类型加到 rcdsub 的实现中。

17.3.3　练习[★★★ ↛]:如果应用规则中的子类型检查失败，类型检查器所打印的错误信息对于用户来说不太有用。我们可以在错误信息中加上期望的参数类型和实际变量类型，但就算这样也不易理解。例如，如果期望的类型是：

```
{x:{},y:{},z:{},a:{},b:{},c:{},d:{},e:{},f:{},g:{}}
```

而实际的类型为：

```
{y:{},z:{},f:{},a:{},x:{},i:{},b:{},e:{},g:{},c:{},h:{}}
```

第二个类型丢失了一个 d 域，但这不能明显看出来。如果改变 subtype 函数，而不是简单返回 true 或者 false，能够使错误报告有很大改进，即返回一个不重要的值[单元值()]或者提示一个异常。因为提示异常的位置正好是类型匹配失败的地方，这样就能较为精确地指出问题的所在。要注意的是，这种改变不影响检查器的“端到端”行为：如果子类型的检查器返回 false，那么类型检查器总会提示异常(通过调用 error)。

重复实现 typeof 和 subtype 函数，可使提示的错误信息尽可能详细。

17.3.4　练习[★★★ ↛]:在 15.6 节中，通过将类型化和子类型化的推导翻译为纯简单类型化 λ 演算的项，为具有记录和子类型的语言定义了强制语义。请修改上面的 subtype 函数来实现这些转换，以便它构造并且返回强制类型函数(用 term 表示)。类似地，修改 typeof 函数，使它返回一种类型和一个被翻译的项。然后，对翻译的项(不是开始输入的项)进行求值，并照常输出结果。

第 18 章　实例分析:命令式对象[①]

从本章开始,我们将接触到第一个实例。我们将用到前面所定义的大部分性质(函数、记录、一般递归、可变引用和子类型化)来建立一个支持类似于 Smalltalk 和 Java 这样的面向对象的程序设计语言中的对象和类的编程模式。本章并不介绍关于对象和类的任何新的具体语法,而是要分析如何使用较低层的构造来模仿复杂语言的行为,以此来弄明白它们的语言特征。

对本章大部分内容,模仿程度非常准确:若把对象和类视为已有特征的简单组合形式,我们能获得对象和类的大部分特征令人满意的实现。当接触到虚函数和 self(参见 18.9 节)时,还会遇到求值顺序上的困难,使这种简化变得有点不现实。若按第 19 章的做法,直接对这些特征的语法、操作语义和类型化规则公理化,则可以得到一个较为满意的结果。

18.1　什么是面向对象编程

大多数关于"某某的精髓是什么"的争论更多暴露出了参与者的偏见,而不是揭露争论话题的客观事实。试图精确地定义"面向对象"也逃不出上述现象。虽然如此,我们还是可以定义一些大多数面向对象语言基本的特征,让它们一起来说明其优缺点和程序设计的风格:

1. **多重表示**:也许面向对象风格的最基本的特性就是,当一个对象调用一个操作时,对象自己确定哪些代码被执行。对相同操作做出反应的两个对象(就是有着相同的接口)可能有着完全不同的表式形式,只要每个的操作实现是根据自己的特殊表示形式进行的就可以了。这种实现称为对象的方法。在对象上调用一种操作(称为方法调用,或者更生动一点,发送一个消息)是在运行时在对象的方法表中查找该操作的名称,这个过程可叫做动态调用。

   比较起来,传统的抽象数据类型(ADT)由一组值及对这组值的单一操作组成(这种静态的定义对对象来说既有优点也有缺点;我们将在 24.2 节更深入地阐述这一点)。
2. **封装**:从对象定义的外部来看,对象的内部表示通常是隐藏的:只有对象自己的方法才能访问或操作它的内部数据[②]。这意味着对象的内部表示的修改只会影响程序的很

---

① 本章以具有子类型化(参见图 15.1)、记录(参见图 15.3)和引用(参见图 13.1)的简单类型 λ 演算的项为例子。相关的 OCaml 实现是 fullref。

② 在一些面向对象语言中,如 Smalltalk,这种封装性是强制性的——对象的内部字段不能在它的定义之外命名。其他的一些语言,例如 C++ 和 Java,允许这些字段被标记为 public 或者 private。相反的是,在 Smalltalk 中,一个对象的所有的方法都可公共地访问,但是 Java 和 C++ 允许方法被标记为 private,只让同一对象上其他方法对它们调用。这里将忽略掉这些精确的说明,但它们已经在一些研究著作中被详细地考虑到了(Pierce 和 Turner, 1993; Fisher 和 Mitchell, 1998; Fisher, 1996a; Fisher 和 Mitchell, 1996; Fisher, 1996b; Fisher 和 Reppy, 1999)。

虽然大多数面向对象语言都把封装作为一个基本的概念,但也有少数的语言并不这样做。多重方法在 CLOS (Bobrow, DeMichiel, Gabriel, Keene, Kiczales 和 Moon, 1988; Kiczales, des Riviéres 和 Bobrow, 1991), Cecil (Chambers, 1992, 1993), Dylan (Feinberg, Keene, Mathews 和 Withington, 1997; Shalit), KEA (Mugridge, Hamer 和 Hosking, 1991) 及 Castagna, Ghelli 和 Longo (1995; Castagna, 1997) 的 λ-& 演算中都有介绍,它是通过在方法调用时使用特殊的类型标记从重载的方法中选择恰当的方法以保持对象状态与方法分离。这些语言在对象构造、方法调用、类定义,等等中的潜在机制从本质上区别于本章中提到的语言,虽然它们产生的高级程序用法是相似的。

小、易于识别的区域;这种约束极大地改进了大型系统的可读性和可维护性。

抽象数据类型提供了相似的封装,确保了其值的具体表示只能在特定的区域内是可见的(例如,一个模块或一个抽象定义),该区域外的代码只能通过调用这些特定区域内定义过的方法来操作这些值。

3. **子类型化**:一个对象的类型(它的接口)是它操作的名字和类型的集合。对象的内部表示不会出现在它的类型中,因为它并不影响我们所要进行的对象操作。

对象接口自然适合于子类型关系。如果一个对象能满足一个接口 I,那么它也将能满足于任何操作方法少于 I 的接口 J,因为任何期望 J 对象的上下文只能调用该对象的 J 方法,所以提供 I 对象总是安全的(这样,对象子类型化就类似于记录的子类型化。实际上,本章中展开的对象模型是相同的)。忽略对象接口的一部分,这样只要编写一段代码并在代码中包括一些公共的功能以统一的方式操作不同的对象。

4. **继承**:共享部分接口的对象也会共享一些行为,但对公共的行为通常只是执行一次。大多数面向对象的语言通过一种称为类的结构(对象实例化的模板)和一种允许继承父类并增加新方法来达到新的实现方式及(必要时)有选择地重载旧方法的继承机制,从而达到重用目的(一些面向对象的语言使用一种称为委托机制来代替类机制,这种机制结合了对象和类的特征)。

5. **开放递归**:大多数具有对象和类的语言提供的另一种易于使用的特征,这就是一个方法能够通过一种特殊变量 self 或在某些语言中的 this 变量,来调用同一对象中的其他的方法。self 的这种特殊的行为是后联编的,即允许一个类中定义的方法去调用较晚(在第一个子类中)定义的方法。

本章接下来的部分继续讨论这些特征,将从非常简单的"独立"对象开始,然后慢慢考虑类的更强功能形式。

随后的章节会检查对象和类不同的特性。第 19 章会以 Java 风格给出对象和类的一个直接处理方法(并非编码形式)。而第 27 章将会回到本章所讨论的编码形式,用囿量词来改进类构造的运行效率。第 32 章将探讨一个运行在纯函数设置中的更有前途的编码形式。

18.2 对象

在对象的最简单的形式中,它只是一个封装一些内部数据并提供外部访问这些数据方法集的数据结构。内部数据由一些可变的实例变量(或字段)组成,这些实例变量是方法共享的,但不能被程序的其他部分所访问。

本章使用的例子都是由对象表示的简单计数器。每一个计数器有一个数值和两个方法(例如,响应两个消息):一个是实现取值的 get 方法,一个是使值增加的 inc 方法。

实现前面章节介绍的特征直接方式是对内部数据使用一个引用单元和对方法使用一个函数记录。一个当前数据为 1 的计数器对象类似于:

```
c = let x = ref 1 in
      {get = λ_:Unit. !x,
       inc = λ_:Unit. x:=succ(!x)};
▸ c : {get:Unit→Nat, inc:Unit→Unit}
```

两个方法体都写成带有平凡参数的函数(用“_”表示不必在程序中指明它们)。当对象被创建时,抽象会阻止方法体的求值,随后允许方法体能够被多次重复求值,将其反复地应用于 unit 参数。并且,注意该对象的状态是如何在方法中共享且不能被程序的其他部分访问的:数据的封装直接源于变量 x 的语义范围。

为调用对象 c 中的一个方法,我们只抽取记录的一个字段并将其应用于一个合适的参数。例如:

```
  c.inc unit;
▸ unit : Unit
  c.get unit;
▸ 2 : Nat
  (c.inc unit; c.inc unit; c.get unit);
▸ 4 : Nat
```

事实是 inc 方法返回了 unit,允许我们在增量的次序上使用“ ;”符号(参见第 11.3 节)。可以将上面的最后一行代码等价地写为:

```
let _ = c.inc unit in let _ = c.inc unit in c.get unit;
```

因为要创建并操作许多计数器,所以采用以下类型的缩写形式将会更方便:

```
Counter = {get:Unit→Nat, inc:Unit→Unit};
```

本章关注的焦点是对象如何被创建,而不是在组织大型的程序中如何使用它们。但是,我们至少想看一下一个使用对象的函数,以证实它是在具有不同内部表示的对象上起作用。下面是一个小例子——一个使用计数器对象并三次调用 inc 方法的函数:

```
  inc3 = λc:Counter. (c.inc unit; c.inc unit; c.inc unit);
▸ inc3 : Counter → Unit
  (inc3 c; c.get unit);
▸ 7 : Nat
```

18.3 对象生成器

我们已经看到如何一次创建一个独立的计数器对象。编写一个计数器的生成器,即一个每次被调用时会创建并返回一个新的计数器对象的函数,同样是简单的事:

```
  newCounter =
    λ_:Unit. let x = ref 1 in
               {get = λ_:Unit. !x,
                inc = λ_:Unit. x:=succ(!x)};
▸ newCounter : Unit → Counter
```

18.4　子类型化

面向对象语言盛行的一个原因就是它们允许同一段代码操作多种形态的对象。例如,假设除了前面定义的 Counter 对象外,还创建了一些对象,含有让它们在任何时候重设初始状态的方法:

```
ResetCounter = {get:Unit→Nat, inc:Unit→Unit, reset:Unit→Unit};
newResetCounter =
  λ_:Unit. let x = ref 1 in
             {get   = λ_:Unit. !x,
              inc   = λ_:Unit. x:=succ(!x),
              reset = λ_:Unit. x:=1};
```

```
▸ newResetCounter : Unit → ResetCounter
```

由于 ResetCounter 含有 Counter 的所有字段(比其还多一个),记录子类型化规则说明:ResetCounter <: Counter 。这说明客户端函数如 inc3(把计数器对象当做参数),能安全地被用于重置计算器对象:

```
rc = newResetCounter unit;
```

```
▸ rc : ResetCounter
```

```
(inc3 rc; rc.reset unit; inc3 rc; rc.get unit);
```

```
▸ 4 : Nat
```

18.5　聚集实例变量

到目前为止,所描述的对象状态都是由一个单独的引用单元组成的。显然,更有趣的对象总是有着不同的实例变量。在接下来的内容里,将所有这些实例变量变成一个独立的单元是十分有用的。为了达到这一目的,让我们将计数器对象的内部表示改成一个引用单元的记录,并且通过映射该记录的字段指向方法体中的实例变量:

```
c = let r = {x=ref 1} in
      {get = λ_:Unit. !(r.x),
       inc = λ_:Unit. r.x:=succ(!(r.x))};
```

```
▸ c : Counter
```

这种实例变量记录的类型称为对象的表示类型:

```
CounterRep = {x: Ref Nat};
```

18.6　简单类

除了 reset 方法外,newCounter 和 newResetCounter 在随后的定义中是相同的。当然,这两种定义都非常简略,以至于使它们几乎没有差别,但如果它们占多页篇幅,如实际可能发生的那样,我们更希望有什么方法能在一个地方描述它们共同的功能。这种在大多数面向对象语言中实现的机制就称为类。

在现实的面向对象语言中的类机制往往是复杂的，并且装载了各种特征——self，super，可见的注释，静态字段和方法，内部类，友元类，像 final 和 Serializable 的注释，等等[①]。我们在此将忽略其中的大部分而把注意力放在类型最基本的特征上：以继承方式的代码重用 self 的后联编。不过此时，只考虑前者。

一个类的最初形式只是一个含方法集合的数据结构，它可以实例化为一个对象，也可以扩展为另一个类。

为什么我们不能从计数器对象中重用一些方法，创建一个重置计数器呢？仅仅因为，在一些特殊的计数器对象中，方法体包含了指向实例变量的一些特殊记录的引用。所以假如我们要对实例变量的不同记录重用相同的代码，需要做的就是抽象关于实例变量的方法。这就等于把上面的 newCounter 函数分成两部分，其中一个定义处理实例变量的任意记录的方法：

```
counterClass =
  λr:CounterRep.
    {get = λ_:Unit. !(r.x),
     inc = λ_:Unit. r.x:=succ(!(r.x))};
▸ counterClass : CounterRep → Counter
```

另一个分配实例变量的一个记录提供给方法以创建一个对象：

```
▸ newCounter : Unit → Counter
  newCounter =
    λ_:Unit. let r = {x=ref 1} in
                 counterClass r;
```

counterClass 中的方法还可以用来定义新的类，称为子类。比如，可以定义一个重置计数器类：

```
resetCounterClass =
  λr:CounterRep.
    let super = counterClass r in
      {get   = super.get,
       inc   = super.inc,
       reset = λ_:Unit. r.x:=1};
▸ resetCounterClass : CounterRep → ResetCounter
```

和 counterClass 类似，这个函数使用一个实例变量的记录，返回一个对象。在内部，它首先用 counterClass 创建了带相同实例变量记录的计算器对象 r；"父对象"用 super 变量表示。然后复制 super 中的 get 和 inc 字段，并为 reset 字段提供一种新的函数来创建一个新对象。由于 super 建立在 r 之上，所以三个方法共享相同的实例变量。

要创建一个重置计数器对象，我们又一次为它的实例变量分配内存，调用 resetCounterClass，实际的工作就在这里运行：

```
newResetCounter =
  λ_:Unit. let r = {x=ref 1} in resetCounterClass r;
▸ newResetCounter : Unit → ResetCounter
```

---

① 所有的这些复杂性的主要原因就是，在大多数这类语言中，类是惟一的大规模结构化机制。实际上，只有一种已经被广泛应用的语言(OCaml)提供了类和复杂的模块系统。所以多数语言中的类趋向于成为堆放所有与大规模程序结构有关的语言特征的垃圾场。

18.6.1　练习[推荐,★★]：写出一个 resetCounterClass 的子类，且该子类含有将 Counter 中的值减 1 的方法 dec。利用 fullref 测试程序测试你的新类。

18.6.2　练习[★★ ↛]：直接把父类的大多数字段复制到子类的方法是相当笨拙的——它虽然避免了在子类中重新输入父类方法的代码过程，但它仍然涉及到很多类型化问题。如果仍用这种风格来开发较大的面向对象程序，希望有一种更简洁的符号出现，比如用"super with { reset = λ _ : Unit.r.x : = 1 }，"表示"一种类似 super 的记录，但有一个含 λ _ : Unit.r.x: = 1函数的字段 reset"。写出这种结构的语法，操作语义和类型规则。

我们应该强调这些类是值，而不是类型。也可建立很多个类，它们能生成相同类型的对象。像 C++ 这样的主流面向对象语言，类还有更复杂的情形——它们既可作为编译时的类型，又可作为运行时的数据结构。这一点在 19.3 节有更深入的讨论。

18.7　添加实例变量

计数器对象和重置计数器对象的内部表示完全相同。但是一般来说，一个子类不仅要扩展父类的继承方法，还要扩充一些继承的实例变量。举个例子，假设定义一个"备份计数器"(backup counters)类，它的 reset 方法把它们的数据重新设置为无论什么值(最后将该方法称为 backup)，而不是把它们重新设置成一个常量：

```
BackupCounter = {get:Unit→Nat, inc:Unit→Unit,
                 reset:Unit→Unit, backup: Unit→Unit};
```

要实现备份计数器，需要另一个实例变量来保存数据的备份值。

```
BackupCounterRep = {x: Ref Nat, b: Ref Nat};
```

如通过复制 get 和 inc 方法并添加 reset 方法将 resetCounterClass 从 counterClass 继承过来，这里也是通过复制 get 和 inc 方法，并给出 reset 和 backup 方法让 backupCounterClass 继承 resetCounterClass：

```
backupCounterClass =
  λr:BackupCounterRep.
    let super = resetCounterClass r in
       {get    = super.get,
        inc    = super.inc,
        reset  = λ_:Unit. r.x:=!(r.b),
        backup = λ_:Unit. r.b:=!(r.x)};

▸ backupCounterClass : BackupCounterRep → BackupCounter
```

这个定义中有两点比较有趣。第一，尽管父类对象 super 含有一个 reset 方法，但还是写了一个新的实现方式，因为我们想得到一个不同的行为。新类重载了父类中的 reset 方法。第二，在类型化表达式(用来建造 super 的)的过程中，子类型起了关键作用：resetCounterClass 需要一个 CounterRep 类型(是参数 r 的实际类型 BackupCounterRep 的超类型)的参数。换句话说，实际上为父对象提供了一个比它的方法所需要的更大的实例变量记录。

18.7.1　练习[推荐,★★]：定义一个 backupCounterClass 的子类，使其含两个新方法，reset2 和 backup2，控制另一个"备份注册"。这个注册应完全独立于 backupCounterClass 所增加的

那一个:调用 reset 时应该保存上次调用 backup 时的值(好像它是刚设的),调用 reset2 时应该保存上次调用 backup2 时的值。用 fullref 检查器检测你的类。

18.8 调用超类方法

变量 super 可用于把函数从父类复制到新的子类。用 super 可以在定义方法时增加新的方法来扩充父类的行为。比如,假设变化一下 backupCounter 类,每调用一次 inc 方法前自动调用一次 backup(天知道这样一个类有什么用处——它仅仅是一个例子):

```
funnyBackupCounterClass =
  λr:BackupCounterRep.
    let super = backupCounterClass r in
       {get = super.get,
        inc = λ_:Unit. (super.backup unit; super.inc unit),
        reset = super.reset,
        backup = super.backup};
```

```
▸ funnyBackupCounterClass : BackupCounterRep → BackupCounter
```

注意这里在 inc 的新的定义中 super.inc 和 super.backup 的调用是怎样避免重复 inc 和 backup 的超类代码。对较大例子,这种情况下保存复制功能有实质性作用。

18.9 含 self 类

最后的扩展是让类中的方法可以通过 self 来相互引用。要激发这个扩展,假设想实现一个 Counter 类,它有一个 set 方法,可从外部把当前的求值设成一个特殊的数:

```
SetCounter = {get:Unit→Nat, set:Nat→Unit, inc:Unit→Unit};
```

而且假设 inc 方法中用到 set 和 get,而不是直接读取和指派实例变量 x(想像一下,一个大型例子中,set 和 get 方法每个定义都需要好几页)。既然 get,set 和 inc 都定义在相同的类中,我们所需要的关键就是使这个类中的方法可以互递归调用。

在 11.11 节提到了怎样使用 fix 操作符来建立函数互递归记录。我们简单的把记录方法抽象为一个参数,这个参数本身是一个函数记录(称为 self),然后使用 fix 操作符"将节点联结",使得建立的记录以 self 形式传递:

```
setCounterClass =
  λr:CounterRep.
    fix
      (λself: SetCounter.
        {get = λ_:Unit. !(r.x),
         set = λi:Nat.  r.x:=i,
         inc = λ_:Unit. self.set (succ (self.get unit))});
```

```
▸ setCounterClass : CounterRep → SetCounter
```

这个类没有父类,因此这里不需要 super 变量。相反,传递的记录以 self 形式传给 inc 方法时,inc 先激活 get,然后激活 set。fix 的使用完全在 setCounterClass 类内部。然后按平时一样的方法创建 set-counter:

```
newSetCounter =
  λ_:Unit. let r = {x=ref 1} in
                setCounterClass r;
```

```
▸ newSetCounter : Unit → SetCounter
```

18.10　使用 self 的开放递归

大多数面向对象的语言实际上支持方法之间更一般的递归形式,称为开放递归,或 self 的后联编。从类定义中去掉 fix 来实现更一般的行为:

```
setCounterClass =
  λr:CounterRep.
     λself: SetCounter.
       {get = λ_:Unit. !(r.x),
        set = λi:Nat.  r.x:=i,
        inc = λ_:Unit. self.set (succ(self.get unit))};
```

```
▸ setCounterClass : CounterRep → SetCounter → SetCounter
```

然后把它放到对象生成函数里:

```
newSetCounter =
  λ_:Unit. let r = {x=ref 1} in
                fix (setCounterClass r);
▸ newSetCounter : Unit → SetCounter
```

注意,移走 fix 后改变了 setCounterClass 的类型:它不仅抽象为一个实例变量的记录,也抽象为 self 对象;实例化时两者都产生实例。

为什么通过 self 的开放递归是有用的,是因为它允许父类方法去调用子类方法,即使父类定义时,子类还不存在。实际上,我们已经改变了对 self 的解释,它不再是"这个类的方法",而是提供了一个访问"已经实例化的对象类(可能为该类的子类)的方法"的方式。

比如,假设要创建 set-counter 的子类,它记录 set 方法被调用的次数。这个类的接口包括一个额外的提取访问次数的操作:

```
InstrCounter = {get:Unit→Nat, set:Nat→Unit,
                inc:Unit→Unit, accesses:Unit→Nat};
```

且表达中包含一个访问次数的实例变量:

```
InstrCounterRep = {x: Ref Nat, a: Ref Nat};
```

在计数器(instrumented counter)类的定义中,inc 和 get 方法从上面定义的 setCounterClass 中复制而来。accesses 方法按一般的方式写出来。在 set 方法中,我们首先让访问数加一,然后使用 super 来激发父类的 set:

```
instrCounterClass =
  λr:InstrCounterRep.
     λself: InstrCounter.
       let super = setCounterClass r self in
         {get = super.get,
          set = λi:Nat. (r.a:=succ(!(r.a)); super.set i),
          inc = super.inc,
          accesses = λ_:Unit. !(r.a)};
▸ instrCounterClass : InstrCounterRep →
                         InstrCounter → InstrCounter
```

因为通过 self 的开放递归,从 inc 的方法体中调用 set 将导致实例变量 a 递增,即使 set 的递增行为在子类中定义,inc 的定义出现在父类中。

18.11 开放递归及求值顺序

在 instrCounterClass 的定义中有一个问题——我们不能用它来创建实例!如果按平常的方法来写 newInstrCounter:

```
λ_:Unit. let r = {x=ref 1, a=ref 0} in
                 fix (instrCounterClass r);

▸ newInstrCounter : Unit → InstrCounter
```

然后把它用到 unit 来试图产生一个计数器:

```
ic = newInstrCounter unit;
```

这个求值将会产生歧义。来看一下它是怎样产生的,从该项开始考虑求值步骤的顺序:

1. 首先把 newInstrCounter 应用到 unit,产生:

   ```
   let r = {x=ref 1, a=ref 0} in fix (instrCounterClass r)
   ```

2. 接着分配两个 ref 单元,把它们包装到一个记录(称为 <ivars>),在后面的部分用 <ivars> 来代替 r:

   ```
   fix (instrCounterClass <ivars>)
   ```

3. 把 <ivars> 传递给 instrCounterClass。由于 instrCounterClass 从两个 λ 抽象开始,我们立即返回到正在等待 self 的函数:

   ```
   fix (λself:InstrCounter.
          let super = setCounterClass <ivars> self in <imethods>)
   ```

   这里 <imethods> 是计数器的方法记录。称这个函数为 <f>,写出目前的状态(fix <f>)。

4. 把这个求值规则应用到 fix(参见图 11.12 中的 E-Fix),"打开"fix <f>,在 <f> 中用(fix <f>)代替 self,产生:

   ```
   let super = setCounterClass <ivars> (fix <f>) in <imethods>
   ```

5. 现在还原 setCounterClass 的应用 <ivars>,产生:

   ```
   let super = (λself:SetCounter. <smethods>) (fix <f>)
               in <imethods>
   ```

   其中 <smethods> 是 set-counter 方法的记录。

6. 根据应用的求值规则,直到(fix <f>)还原为一个值时,才能还原(λself : SetCounter. <smethods>)的应用(fix <f>)。因此求值的下一步是再次打开 fix <f>,得到:

   ```
   let super = (λself:SetCounter. <smethods>)
                 (let super = setCounterClass <ivars> (fix <f>)
                  in <imethods>)
   in <imethods>
   ```

7. 由于外层的 λ 抽象的参数还不是一个值,所以必须继续求值到里边的一层。把 setCounterClass 应用到 <ivars>,得到:

```
let super = (λself:SetCounter. <smethods>)
              (let super = (λself:SetCounter. <smethods>)
                              (fix <f>)
               in <imethods>)
in <imethods>
```

8. 现在创建了一个形式上与外层相似的内层应用。和前面一样，这个内层的应用不能被简化，直到它的参数 fix < f > 被完全求值。因此我们的下一步是再次打开 fix < f >，得到一个形式上和第 6 步一样但更深嵌套的表达式。但在这点上还应该清楚，决不是要去给外层求值。

直观来看，这里的问题是 fix 操作符的参数过早地使用自己的参数 self 。fix 的操作语义是希望把 fix 应用到某个函数 λx.t 时，t 只能处于"受保护"地位指向 x，如里层的 λ 抽象。比如，在 11.11 节定义的 iseven，把 fix 应用到形式为 λie.λx.⋯ 的函数中，其中函数体中对 ie 的递归引用就受到 x 抽象的保护。相反，instrCounterClass 的定义总想计算 super 的值后立即使用 self。

这一点上，可以有多种方式来进行：

- 可以在 instrCounterClass 里保护对 self 的引用，以避免被过早求值，比如插入虚拟 λ 抽象。下面就要讨论这个解决方法。我们将会看到它并不完全让人满意，但它能直观地描述和理解已出现机制的使用方式。第 32 章考虑纯函数对象编码时，会发现它也很有用。
- 可以寻找不同的方式，利用低级语言的特征来构建类的语义模型。比如，不用 fix 来构建类的方法表，而使用引用来更直接地构建。我们将在 18.12 节实现这个想法，在第 27 章还会有更深入的定义。
- 从 λ 抽象、记录和 fix 角度来说，可以先不考虑对象和类的编码，把它们当成语言中带自己求值(或类型)规则的原语。那样就可以简单地选择符合对象和类实现功能意图的求值规则，而不是围绕着所给的应用和 fix 规则带来的问题。这个方法在第 19 章中还会谈到。

使用虚拟 λ 抽象来控制求值顺序在函数编程领域是一个为人熟知的小技巧。其思想是任意一个表达式 t 可以转换成一个函数 λ _: Unit.t，这称为 thunk。t 的"thunk 形式"是一个语法值；所有涉及到求值 t 的计算将推迟，直至转换应用于 unit。这提供了一个把 t 以未赋值的形式传递的方法，可以后再考虑结果。

这时可以推迟 self 的求值。通过把它的类型从一个对象(比如 SetCounter)改成一个 thunk 对象(如 Unit→SetCounter)来实现：包括(1)把 self 参数的类型改为类；(2)在创建最终对象之前增加一个虚拟抽象；(3)在方法体中把所有出现 self 的地方改成(self unit)。

```
setCounterClass =
  λr:CounterRep.
  λself: Unit→SetCounter.
    λ_:Unit.
      {get = λ_:Unit. !(r.x),
       set = λi:Nat.  r.x:=i,
       inc = λ_:Unit. (self unit).set(succ((self unit).get unit))};
▸ setCounterClass : CounterRep →
                    (Unit→SetCounter) → Unit → SetCounter
```

虽然不想改变 newSetCounter 的类型(它仍然返回一个对象),但它的定义还是要稍微修改一下,使得形成 setCounterClass 的不动点时,它可以传递一个 unit 参数给它产生的 thunk:

```
newSetCounter =
  λ_:Unit. let r = {x=ref 1} in
             fix (setCounterClass r) unit;

▸ newSetCounter : Unit → SetCounter
```

需对 instrCounterClass 的定义中做类似修改。注意所有这些修改都不需要任何思考:一旦修改了 self 的类型,所有其他的修改都要根据类型规则的指示:

```
instrCounterClass =
  λr:InstrCounterRep.
  λself: Unit→InstrCounter.
    λ_:Unit.
      let super = setCounterClass r self unit in
         {get = super.get,
          set = λi:Nat. (r.a:=succ(!(r.a)); super.set i),
          inc = super.inc,
          accesses = λ_:Unit. !(r.a)};

▸ instrCounterClass : InstrCounterRep →
                      (Unit→InstrCounter) → Unit → InstrCounter
```

最后修改 newInstrCounter 以便于让它为 fix 创建的 thunk 提供一个虚参:

```
newInstrCounter =
  λ_:Unit. let r = {x=ref 1, a=ref 0} in
             fix (instrCounterClass r) unit;

▸ newInstrCounter : Unit → InstrCounter
```

现在可以用 newInstrCounter 来创建一个实际的对象了:

```
ic = newInstrCounter unit;

▸ ic : InstrCounter
```

回想一下,这就是我们添加 thunk 之前发散的那一步。

下面的测试描述了 accesses 方法怎样计算调用 set 和 inc 的次数:

```
(ic.set 5; ic.accesses unit);

▸ 1 : Nat

(ic.inc unit; ic.get unit);

▸ 6 : Nat

ic.accesses unit;

▸ 2 : Nat
```

18.11.1 练习[推荐,★★★]:利用 fullref 检查器来为上面的类实现下面扩充:

1. 重写 instrCounterClass,以使它能记录调用 get 的次数。

2. 用一个子类来扩充你所修改的 instrCounterClass,像 18.4 节一样,子类里增加一个 reset 方法。
3. 再增加一个子类,支持备份,像 18.7 节一样。

18.12　更高效的实现

上面的测试显示类的实现符合 Smalltalk,C++ 和 Java 语言中通过 self 的方法调用的"开放递归"行为模式。但是应该注意,这个实现从效率方面来考虑并不完全让人满意。为了使 fix 计算集中一点而增加的 thunk 会推迟类的方法表的计算。特别是,注意在方法体里边对 self 的所有调用都变成了(self unit)——也就是说,每次无意中对它们做递归调用时 self 方法就被重新计算一次!

在创建对象时,使用引用单元而不用类层次结构中不动点来"联结节点",这样可以避免所有的重复计算[①]。不是对类中 self 方法(后来用 fix 创建的)的记录进行抽象,而是抽象一个引用到方法记录,然后先分配这个记录。也就是说,我们建立一个类实例时,首先为它的方法(用一个虚拟值初始化)分配一个堆单元,然后创建实际的方法(给它们传递一个指向堆单元的指针,通过指针可以做递归调用),最后回补堆单元,使它包含实际的方法。例如,又见 setCounterClass:

```
setCounterClass =
  λr:CounterRep. λself: Ref SetCounter.
    {get = λ_:Unit. !(r.x),
     set = λi:Nat. r.x:=i,
     inc = λ_:Unit. (!self).set (succ ((!self).get unit))};
▸ setCounterClass : CounterRep → (Ref SetCounter) → SetCounter
```

参数 self 是一个指向包含当前对象的方法单元的指针。调用 setCounterClass 时,这个单元用一个虚拟值初始化:

```
dummySetCounter =
  {get = λ_:Unit. 0,
   set = λi:Nat.  unit,
   inc = λ_:Unit. unit};
▸ dummySetCounter : SetCounter

newSetCounter =
  λ_:Unit.
    let r = {x=ref 1} in
    let cAux = ref dummySetCounter in
    (cAux := (setCounterClass r cAux); !cAux);
▸ newSetCounter : Unit → SetCounter
```

但是,由于所有的反引用操作(!self)都受到 λ 抽象的保护,所以该单元直到 newSetCounter 回补后才被真正地反引用。

---

① 这本质上和练习 13.5.8 所使用的解决方法是相同的想法。感谢 James Riely,他观察到利用 Source 类型的协变性可以将该想法应用到类结构上。

为了支持创建 setCounterClass 的子类,需要对它的类型进一步精化。每个类都希望它的 self 参数拥有与它创建的方法记录相同的类型。也就是说,如果我们定义一个计数器的子类,这个类的 self 参数将是一个指向计数器方法的记录指针。但是,如在 15.5 节所见,Ref SetCounter 类型和 Ref InstrCounter 类型是不相容的——把后者提升为前者是不可靠的。在 instrCounterClass 的定义中试图创建 super 会引起麻烦(也就是参数类型不匹配):

```
instrCounterClass =
  λr:InstrCounterRep. λself: Ref InstrCounter.
  let super = setCounterClass r self in
  {get = super.get,
   set = λi:Nat. (r.a:=succ(!(r.a)); super.set i),
   inc = super.inc,
   accesses = λ_:Unit. !(r.a)};

▸ Error: parameter type mismatch
```

这个难题的解决办法是用 Source 来代替 self 类型中的 Ref 构造器——也就是仅传递给类从方法指针处读取的功能,而不是写入的功能(它怎么也不会需要这个功能)。如在 15.5 节所见,Source 允许协变的子类型(也就是说,有 Ref InstrCounter <: Ref SetCounter),因此 instrCounterClass 里创建的 super 将变为良类型的:

```
setCounterClass =
  λr:CounterRep. λself: Source SetCounter.
    {get = λ_:Unit. !(r.x),
     set = λi:Nat. r.x:=i,
     inc = λ_:Unit. (!self).set (succ ((!self).get unit))};

▸ setCounterClass : CounterRep → (Source SetCounter) → SetCounter

instrCounterClass =
  λr:InstrCounterRep. λself: Source InstrCounter.
  let super = setCounterClass r self in
  {get = super.get,
   set = λi:Nat. (r.a:=succ(!(r.a)); super.set i),
   inc = super.inc,
   accesses = λ_:Unit. !(r.a)};

▸ instrCounterClass : InstrCounterRep →
                        (Source InstrCounter) → InstrCounter
```

要创建一个计数器对象,像前面一样,首先定义一个计数器方法的虚拟集合,以作为 self 指针的初始值:

```
dummyInstrCounter =
  {get = λ_:Unit. 0,
   set = λi:Nat.  unit,
   inc = λ_:Unit. unit,
   accesses = λ_:Unit. 0};

▸ dummyInstrCounter : InstrCounter
```

然后通过为实例变量和方法分配堆空间来创建一个对象，调用 instrCounterClass 来创建实际的方法，回补引用单元：

```
newInstrCounter =
  λ_:Unit.
    let r = {x=ref 1, a=ref 0} in
    let cAux = ref dummyInstrCounter in
    (cAux := (instrCounterClass r cAux); !cAux);

▸ newInstrCounter : Unit → InstrCounter
```

现在构建方法表的代码(在 instrCounterClass 和 counterClass 里)是创建一个对象时一次生成，而不是方法激活时生成。这实现了想要做的事情，但是仍不如想像中有效：毕竟为每个计数器对象创建的方法表是完全相同的，因此似乎在定义类时应该能一次计算这些方法表，以后不再需要。第 27 章将会介绍如何利用第 26 章介绍的囿量词实现这个想法。

18.13 小结

本章的开头列出了好几条面向对象编程风格的特征。现在让我们来回顾一下这些特征，简单讨论它们是怎样与本章的例子发生联系的。

1. **多态**：本章看到的所有对象都是 counter，也就是说，它们都属于 Counter 类型。但是从 18.2 节的单个引用单元到 18.9 节包含多个引用的记录，它们的表示方式差别很大。每个对象都是一个函数的集合，提供了适合于自己内部的表示方式的 Counter 方法(以及一些可能的其他方法)的实现。
2. **封装**：实际上对象的实例变量只有对象中的方法才能访问，而方法的创建也是通过把实例变量记录传递给构造子来实现的。很明显这些实例变量只能用内部方法来命名。
3. **子类型**：在这个集合中，对象类型之间子类型化也就是函数记录类型之间的一般子类型化。
4. **继承**：通过从已存在的父类里边把方法的实现复制到一个新定义的子类里边来实现继承。这里有一些有趣的技术：严格来说，父类和新的子类都是从实例变量到方法集合的函数。子类等待它的实例变量，然后用给定的实例变量来初始化父类，形成一个操作在相同变量上父类的方法集合。
5. **开放递归**：这里构建了由现实世界面向对象语言中的 self(或 this)提供的开放递归模型，通过在实例变量上和 self 参数上抽象出类，self 可以用于方法中来指向同一个对象中的其他方法。这个参数在对象生成时通过使用 fix 去“联结节点”而得到解决。

18.13.1　练习[★★★]：对象有另一个有用的特征称为对象相等，即 sameObject 操作，如果它的两个参数赋给完全相同的对象，结果为 true，如果被赋值的对象是不同时间内生成的(对 new 函数不同的调用)，返回值为 false。怎样把本章的对象模型进行扩展来支持对象相等？

18.14 注释

对象编码是编程语言研究领域里的主要示例和问题来源。早期的编码由 Reynolds(1978)

给出;Cardelli(1984)的一篇文章激起了这个领域的广泛兴趣。从不动点角度对 self 的理解,Cook(1989),Cook 和 Palsberg(1989),Kamin(1988)以及 Reddy(1988)都有所阐述;Kamin 和 Reddy (1994)以及 Bruce (1991) 发掘了这些模型之间的关系。

这个领域里的大量早期的重要论文都收集在 Gunter 和 Mitchell(1994)里。后面的研究有 Bruce,Cardelli 和 Pierce(1999)以及 Abadi 和 Cardelli(1996)。Bruce(2002)给出了这个领域到目前为止的先进展望。其他的基础方法和它们的类型系统可以在 Palsberg,Schwartzbach(1994)和 Castagna(1997)中找到。

还有另外一些历史记录可以在第 32 章的结尾处找到。

继承得到过高评价。

——Grady Booch

第 19 章　实例分析:轻量级的 Java[①]

在 18 章中介绍了含子类型、记录类型和引用类型的 λ 演算如何对面向对象程序的关键特征进行形式化。那一章的目的是通过对更基本特征进行编码来加深对这些特征的理解。本章将采取不同的方法,说明怎样用前面几章的思想来对一个基于 Java 的核心面向对象语言进行直接处理。先来熟悉一下 Java 语言。

19.1　引言

形式化建模能极大地推进复杂世界人造物品,如程序语言的设计。形式化模型可以精确地描述设计的某些方面,说明和证明它的性质,并直接关注那些易被忽视的焦点。然而,在形式化建模过程中,完备性和紧致性之间有一种矛盾:模型涉及的方面越多,就变得越难处理。通常选择一个不太完备但更紧致的模型,使消耗最小但观察最深刻。这种策略在最近掀起了一阵关于 Java 形式化特征的论文热潮,Java 忽略了一些高级特征,如并发和自反,将注意力放在整个语言的片断上,将易理解的理论应用其中。

轻量级的 Java(简称为 FJ),由 Igarashi,Pierce 和 Wadler(1999) 提出,争取 Java 类型系统建模的内核演算最小化。FJ 的设计对紧致性的支持完全超过了完备性,仅仅有 5 种项的形式:对象创建、方法调用、字段访问、强制转型和变量。它的语法、类型规则和操作语义非常适合写在一页纸上(信纸长度),且设计目标是尽可能地忽略一些特征(即使是分配),只保持 Java 类型的核心特征。FJ 与 Java 的纯函数核心直接对应,在一定意义上每一个 FJ 程序都是可执行的 Java 程序。

FJ 仅仅比 λ 演算或者 Abadi 和 Cardelli 的对象演算(1996)复杂一点,但比其他基于类的语言,如 Java 的形式模型要简单很多,这些形式模型包括由 Drossopoulou, Eisenbach 和 Khurshid(1999),Syme(1997),Nipkow 和 Oheimb(1998),Flatt,Krishnamurthi 和 Felleisen(1998a,1998b)等提出的。正因为简单,FJ 能集中关注一些关键问题。比如,在一小步操作语义中捕捉 Java 的强制转型构造比预想的要复杂。

FJ 的主要应用是对 Java 的扩展建模。由于 FJ 本身就很紧致,它将注意力集中在扩展的基本方面。而且,因为纯 FJ 的类型安全性证明非常简单,对更大的扩展安全性严格证明仍是易处理的。最初 FJ 文章通过用一般的类和方法 à la GJ(Bracha, Odersky, Stoutamire 和 Wadler, 1998)扩充 FJ 的方式说明了它的作用。一篇后续的文章(Igarashi, Pierce 和 Wadler, 2001)形式化了最初的类型,这是 GJ 中的特征,用来缓和 Java 程序到 GJ 的演变。Igarashi 和 Pierce(2000)用 FJ 作为 Java 的内部类特征研究的基础。FJ 也被用来作为 Java 的类型保持编译(League, Trifonov 和 Shao,2001)和语义基础(Studer,2001)来研究。

---

① 本章的例子全是关于轻量级 Java 的(参见图 19.1 到图 19.4)。这里没有相关的 OCaml 实现;因为 FJ 被严格设计为 Java 的子集,所有例子都可以在 Java 上实现。

设计 FJ 的目标是使类型安全性的证明尽可能简明,它仍是为了获得整个 Java 核心特征的安全性实质。那些对证明起不到作用只会增加其长度的特征都可以被省略。如其他此类研究一样,FJ 省略了并发性和自反性这些高级特征。其他在 FJ 中所不含的 Java 特征包括分配、接口、过载、传消息给 super,null 指针、基本类型(int,bool 等)、抽象方法声明、内部类、子类字段对父类字段的覆盖、访问控制(public,private 等)和异常,等等。FJ 要建模的 Java 特征,包括互递归类的定义、对象创建、字段访问、方法调用、方法重载、通过 this 的方法递归、子类型化和强制转型。

在 FJ 中一个关键的简化是省略了分配。假定一个对象的字段由它的构造子初始化了,并且以后一直不会变化。但这限制了 FJ 成为 Java"函数"的片断,使其中很多公用的 Java 术语,比如枚举类型,不能被表示出来。但是,这个片断在计算上是完备的(很容易在编码中加入 λ 演算),并且足够大来包括有用的程序,比如很多在 Felleisen 和 Friedman 的 Java 文章(1998)中的许多程序用的就是纯函数风格。

19.2 概要

在 FJ 中,程序包括类定义集合及一个待求值的项,对应于整个 Java 中的 main 方法。下面是一些 FJ 中典型的类定义:

```
class A extends Object { A() { super(); } }

class B extends Object { B() { super(); } }

class Pair extends Object {
  Object fst;
  Object snd;
  // Constructor:
  Pair(Object fst, Object snd) {
    super(); this.fst=fst; this.snd=snd; }
  // Method definition:
  Pair setfst(Object newfst) {
    return new Pair(newfst, this.snd); } }
```

有了语法规则,要经常将超类(甚至它是 Object)包含进来,并要写出构造子[①] (甚至对不重要的类 A 和类 B),而且要在一个字段访问或方法调用中给接收方命名(如在 this.snd),甚至当接收方是 this 时。构造子总是采用相同风格:每个字段有一个参数,取与字段同样的名字;super 构造子用来初始化超类的字段;剩下的字段被相应的参数初始化。在这个例子中,三个类的超类都不含字段的 Object,因此,super 调用不含参数。在一个 FJ 程序中构造子惟一出现在 super 或者"="出现的地方。FJ 不提供副操作,一个方法体总是包含了 return,并紧跟其后一个项,如 setfst()中所示。

在 FJ 中项有 5 种形式,比如:new A(),new B()和 new Pair(…,…)都是对象构造子,还有….setfst(…)是方法调用。在 setfst 体中,项 this.snd 是字段访问,newfst 和 this 是变量[②]。在上述定义的上下文中,项:

---

① 在 Java 中,一般将 constructor 译为构造方法,但为了与全书统一我们在此仍译为"构造子"——译者注。

② 这里 FJ 对 this 的处理方式与 Java 稍微有些不同,是把它当成变量,而不是关键字。

```
new Pair(new A(), new B()).setfst(new B())
```

求值为 new Pair(new B(),new B())。

项还有一种形式是强制转型(参见 15.5 节)。项：

```
((Pair)new Pair(new Pair(new A(), new B()),
                new A()).fst).snd
```

求值为 new B()。子项(Pair)t 是一个强制转型，其中 t 是 new Pair(…).fst。强制转型是需要的，因为 t 对 fst 字段的一个访问，并且声明包含一个 Object，然而接下来对 snd 字段的访问，仅仅对 Pair 有效。在运行时，求值规则要检查存储在 fst 字段中的 Object 是否是一个 Pair(这时检查成功)。

去掉副作用会产生一个好的作用：在 FJ 的语法中，求值容易完全形式化，不会产生对堆建模的附加机制(参见第 13 章)。存在三个基本计算规则：(1)对字段访问；(2)方法调用；(3)用来强制转换。回想在 λ 演算中，应用的求值规则假设函数是第一次被简化为 λ 抽象。同样，在 FJ 中，求值规则假设上面的对象是第一次被简化为一个 new 项。在 λ 演算中有句话"一切都是函数"，而这里则变为"一切都是对象"。

如下是正在执行的字段访问(E-ProjNew)的规则：

```
new Pair(new A(), new B()).snd   →   new B()
```

由于对象构造子的语法经过风格化，构造子每一字段都有一个参数，它们是按照字段被声明的顺序。这里的字段是 fst 和 snd，对字段 snd 的访问选择了第二个参数。

如下是正在执行的方法调用规则(E-InvkNew)：

```
new Pair(new A(), new B()).setfst(new B())
→  [ newfst ↦ new B(),                     ]
   [ this ↦ new Pair(new A(),new B())      ]
   new Pair(newfst, this.snd)
   i.e.,  new Pair(new B(), new Pair(new A(), new B()).snd)
```

被调用的对象因为是 new Pair(new A(),new B())，所以先在类 Pair 中查找 setfst 方法，发现它有形参 newfst 和方法体 new Pair(newfst,this.snd)。实现过程是将实参取代形参，并用目标对象取代 this 变量。这有些类似于 λ 演算的 β 归约规则(E-AppAbs)。关键的差异是要由被调用方的类来决定到何处寻找方法体(支持方法重载)及被调用方对 this 的代换(支持"通过 self 的开放递归")[①]。在 FJ 中，与 λ 演算一样，如果形参在方法体中不止一次地出现，将会导致参数值复制，但因为不会产生副作用，所以看不出与标准 Java 语义的差别。

这是正在执行的强制转换规则(E-CastNew)：

```
(Pair)new Pair(new A(), new B())  →  new Pair(new A(), new B())
```

一旦强制转换的目标归约成了一个对象，就容易看出其构造子的类是转型目标的子类。如果是这样(这里所说的情况)，归约后会去掉强制转换符。如果不是，如项(A) new B()，没有规则可以应用，计算无法进行下去，会显示一个运行错误。

① 熟悉 Abadi 和 Cardelli 的对象演算(1996)的读者将发现这与他们的ς归约规则十分相似。

有三种情况使计算无法进行:(1)试图访问类中没有声明的字段;(2)调用类中没有声明的方法("消息无法理解");(3)试图将对象执行时的类强制转换为除其超类外的别的类。我们将会证明前两种情况在良类型程序中绝不会发生,第三种情况在不含向下转型[及无"愚蠢转型"(一个技术性问题,下面会进一步解释)]的良类型程序中绝不会发生。

这里采用标准的值调用求值策略。以下是对上面第二个例子的项进行求值的过程,每步都用下划线标注了将被归约的下一个子项:

```
   ((Pair) (new Pair(new Pair(new A(),new B()), new A())
                .fst).snd
→  ((Pair)new Pair(new A(),new B())).snd
→  new Pair(new A(), new B()).snd
→  new B()
```

19.3 规范化和结构化的类型系统

在继续 FJ 的形式化定义之前,应该停下来检查一下 FJ(和 Java)与类型化 λ 演算(本书的重点)之间基本的风格差异。这个差异涉及到类型名字的状态。

在前面的章节中,我们经常给长的或者复杂的复合类型定义短的名字来提高例子的可读性,如:

```
NatPair = {fst:Nat, snd:Nat};
```

这种定义是肤浅的:NatPair 的名字是{fst:Nat,snd:Nat}的一个简单的缩写,这两个在上下文中是可以互换的。我们对演算的形式表示已经不考虑缩写了。

相比之下,和许多广泛应用的程序语言一样,Java 的类型定义起着十分重要的作用。Java 中每个复合类型都有一个名字,并且,在定义一个局部变量,字段或方法参数的类型时,都要给一个名字。"裸露的"类型名像{fst:Nat,snd:Nat}这时是不能出现的。

而且这些类型名在 Java 子类型关系中也起着至关重要的作用。无论何时引入一个新名字(在类或者接口定义),程序员都要明确地声明新名字是从哪一个类或接口扩展出来的(或实现的是哪些新类和已经存在的接口)。编译器检查这些声明,以确保新类或接口提供的功能确实是对每个父类和父接口提供功能的扩展——这种检查对应于类型化 λ 演算中的记录子类型化。子类型关系是在类型名字之间,作为声明的直接子类型关系的自反和传递闭包而定义的。如果一个名字没有被说明为另一个子类型,那它就不是名字。

像 Java 这样(名字起重要作用且子类型也明确地声明)的类型系统称为规范化的。而本书中提到的大部分类型系统(名字无关紧要,子类型在类型结构上被直接定义)称为结构化的。

建立在结构化表示上的规范化类型系统既有优点又有缺点。最重要的优点是规范化系统的类型名称不仅在类型检查中起作用,而且在运行时同样起作用。大部分规范化语言给每个运行的对象加上一个含它的类型名称的字头标记,作为一个指针指向运行时的数据结构(该数据结构描述了类型并含有一个指向直接父类的指针)。这些类型标记作用很广,包括运行类型测试(比如 Java 的 instanceOf 测试和向下转型操作)、打印、将数据结构整理为二进制形式,以便在文件中存储或通过网络传输,以及提供一个允许程序动态地检查对象本身的字段和方法的自反手段。运行时类型标记也能被结构化系统支持(参见 Glew,1999;League,Shao 和 Trifonov,

1999;League,Trifonov 和 Shao,2001;以及其中给出的引文),但是它们形成了一个附加的、独立的机制;在规范化系统中,运行时标记就被当成了编译时类型。

规范系统中一个不算基本但比较令人满意的性质是它们提供一个自然并直观的概念:递归类型——类型的定义中提到了它自己(第 20 章将详细讨论递归类型)。这种类型在重要的程序设计中是普适的,要求用来描述一些通用的结构,如列表和树,且规范类型系统直接支持这些结构:指向自身声明体中 List 与指向别的类型一样容易。的确,即使互递归类型也是直接的。我们认为类型名称是一开始就给定的,所以,如果类型 A 的定义中包含 B 的名称,而 B 的定义又指向了 A,那么究竟“哪个先定义”都没有关系。当然,结构化类型系统中也能有递归类型。确实,带结构类型的高级语言,如 ML 语言,通常是将递归类型与其他的特征“捆绑”在一起的,所以对程序员来说,使用时的感觉和规范系统中一样自然和简单。但某些更基本的演算,如类型安全性证明,要求严密地处理递归类型的机制会变得更复杂些,尤其是允许互递归的情况。在规范系统中可以自由使用递归类型确实带来了很大好处。

规范系统另外一个优点是检查一个类型是否是另一个类型的子类型几乎变得没有必要了。当然,编译器仍要验证声明的子类型关系是否安全,这本质上复制了结构化子类型关系,但该项工作只能在类型定义时一次完成,而不是在每次子类型检查时进行的。这使得规范化类型系统的类型检查器多少容易达到一个好的性能。但在更复杂的编译器中,规范化和结构化风格的差异对性能的影响有多大还不太清楚,因为结构化系统中设计良好的类型检查器包括了表示上的一些技巧,这样也能将大部分的子类型检查简化为一个单独的比较(参见 17.2 节)。

还有一个经常被子类型声明引用的优点就是它能阻止“假包含”,发生假包含时类型检查器接受了一段本该使用某个类型但实际却使用了另一个完全不同,但结构相容的类型程序。这一点比上面所述的更有争议,因为存在其他(更好论证的)方法来阻止假包含,例如,使用单结构数据类型(参见 11.10 节)或抽象数据类型(参见第 24 章)。

有了这些优点(尤其是类型标记的有效性及递归类型带来的简单处理),规范类型系统成为主流程序设计语言的标准并不奇怪。另一方面,有关程序设计语言的研究著作几乎全都涉及了结构化类型系统。

造成这些的一个直接原因是至少没有了递归类型,结构化系统会显得更有条理和更加高雅。在结构化设置中,类型表达是闭的实体:它含有所有可以理解它的信息。而规范化系统,总是要根据类型名称的全局性和相关定义来理解类型。这使得定义和证明都很冗长。

一个更有意思的原因是这些研究专著倾向于关注更加高级的特征(尤其是关于类型抽象的机制,如参数多态性,抽象数据类型,用户定义的类型算子和算符等),以及关注包含这些特征的 ML 和 Haskell 等语言。可惜的是,这些特性不太适合规范化系统。例如,类型 List(T),像是不可约简的复合形式——在程序中构造子 List 在某处只有一种定义,而且还要根据 List(T) 功能判断它的定义形式,所以不能把 List(T)简单当成一个原子名称。一些规范化语言已扩展为带这些“一般”的特征(参见 Myers,Bank 和 Liskov,1997;Agesen,Freund 和 Mitchell,1997;Bracha,Odersky,Stoutamire 和 Wadler,1998;Cartwright 和 Steele,1998;Stroustrup,1997),但这种扩展产生的不再是纯规范化系统,而是两种方法的复杂混合。所以带高级类型化特征的语言的设计者们更偏爱结构化方法。

规范化和结构化系统之间关系还有待继续深入研究。

19.4 定义

从现在开始提出 FJ 形式化定义。

语法

图 19.1 给出了 FJ 的语法。元变量 A,B,C,D 和 E 为类名;f 和 g 为字段名;m 为方法名;x 表示参数;s 和 t 表示项;u 和 v 表示值;CL 表示类声明;K 表示构造子声明;M 表示方法声明。假定变量集中含特殊的变量 this,但 this 不能作为传递给方法的变量名。但可以认为 this 隐含在所有的方法声明中。进行方法调用的求值规则(参见图 19.3 的 E-InvkNew 规则)除了用参数值代换参数外,还会用一个合适的对象代换 this。

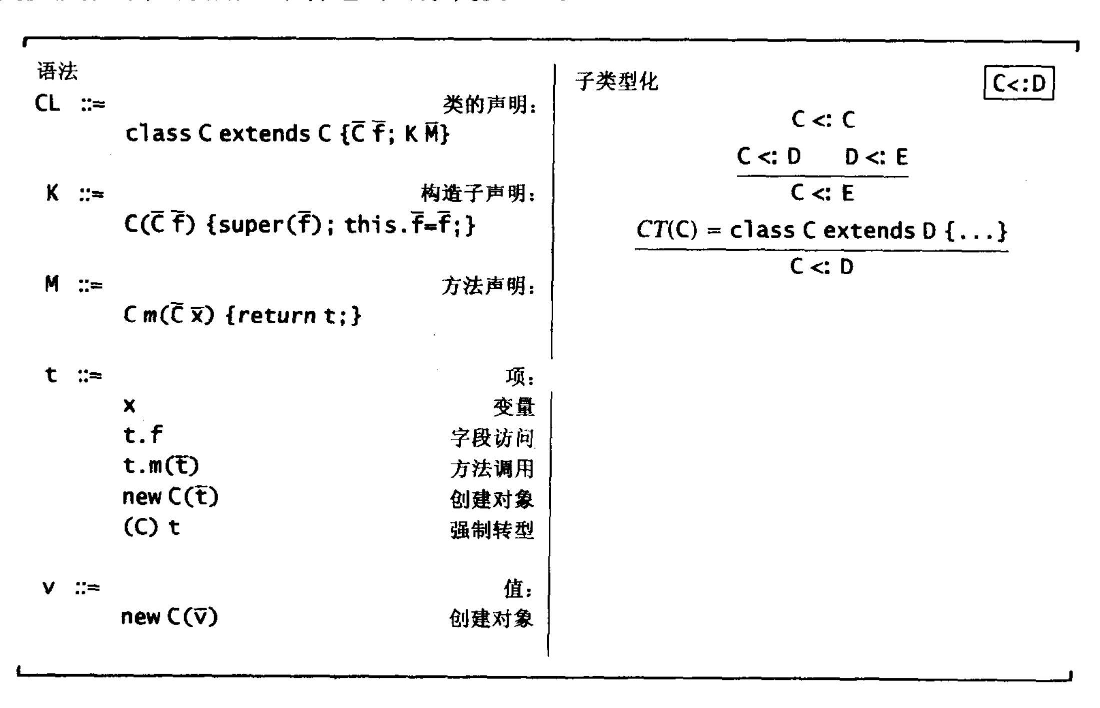


图 19.1 轻量级 Java(语法和子类型化)

用 $\bar{f}$ 表示 $f_1,\cdots,f_n$ 的缩写($\bar{C}$,$\bar{x}$,$\bar{t}$ 也是类似含义等),用 $\bar{M}$ 表示 $M_1\cdots M_n$)(没有逗号)。用同样方法简写系列序对的操作,如用$\bar{C}$ $\bar{f}$表示 $C_1 f_1 \cdots C_n f_n$,其中 n 是 $\bar{C}$ 和 $\bar{f}$ 的长度,且"this.$\bar{f}$ = $\bar{f}$;"表示"this.f_1 = f_1;…;this.f_n = f_n;"。字段声明、参数名、方法声明等系列中都假定不包含重复的名字。

声明 class C extends D{$\bar{C}$ $\bar{f}$;K $\bar{M}$}表示名字为 C 的类有超类 D。新类含有类型为 $\bar{C}$ 的字段 $\bar{f}$,一个构造子 K 和一组方法 $\bar{M}$。由 C 声明的实例变量加到了由 D 声明的变量和它的超类中,所以应该将这些名字区分开①。另一方面,C 的方法将重载 D 中同名的方法或给 C 添加新功能。

① 在 Java 中,超类的实例变量将会被重新声明一次,且新声明的变量会覆盖当前类及其子类的原始变量。FJ 忽略了该特征。

构造子声明 $C(\overline{D}\ \overline{g}, \overline{C}\ \overline{f})$ {super($\overline{g}$);this.$\overline{f}$ = $\overline{f}$;} 表明了如何初始化 C 的实例字段。它的形式完全由 C 的实例变量声明和它的超类决定:它尽可能取得和实例参数一样多的参数,并且函数体内必须包含调用超类构造子来初始化的字段,且将参数指派给 C 声明中同名的新字段(这些限制可由类的类型化规则执行,参见图 19.4)。在整个 Java 中,子类构造子必须包含一个对超类构造子的调用(当超类构造子带参数时);所以存在这里构造子调用 super 来初始化超类的实例变量。如果不考虑将 FJ 作为 Java 的字面子集,可以删除对 super 的调用而直接用每个构造子来初始化所有的实例变量。

方法声明 D m($\overline{C}\ \overline{x}$){return t;} 介绍了 m 方法的结果类型为 D 且含类型为 $\overline{C}$ 的参数 $\overline{x}$。方法体含一条语句 return t。变量由 t 界定。同样特殊变量 this 也认为由 t 界定。

类表 *CT* 是一个从类名 C 到类定义 CL 的映射。一个程序就是类表和项的序对(*CT*,t)。为了减少符号量,我们总是假定一个固定的类表 *CT*。

每个类都有一个超类,用 extends 声明。于是引发了一个问题:Object 类的超类是什么? 有多种方法来处理这个问题;最简单的(这里采用的)是将 Object 作为一个其定义不出现在类表中的特殊类名。对在类表中查找字段的附加功能,如果查找的是 Object,则返回一个空字段,用"•"表示;对象被假定不含方法[①]。

看类表时,可以从类间读出子类型关系。用 C <: D 表示 C 是 D 的一个子类型,也就是说,子类型 *CT* 中的 extends 子句给出的直接子类关系的自反和传递的闭包关系。图 19.1 中给出了正式定义。

给出的类表被认为是满足一些可靠条件:(1) *CT*(C) = class C…对每个 C∈ *dom*(*CT*);(2)Object ∉ *dom*(*CT*);(3)每个类名 C(除了 Object)出现在 *CT* 中任何地方,有 C∈ *dom*(*CT*);(4)在 *CT* 中不含任何循环的子类型关系——也就是"<:"关系是反对称的。

注意被类表定义的类型允许递归,即类 A 的定义中用到的方法类型和实例变量可能包含了名字 A。类间互递归是允许的。

19.4.1　练习[★]:用 λ 演算中的 S-Top 规则与子类型进行类比,可能会发现 Object 是所有类的超类这条规则,但这里为什么不需要它呢?

辅助定义

对于类型化和求值规则,还需要一些辅助的定义(图 19.2 已经给出)。类 C 的字段,记为 field(C),对类 C 和它的所有超类中所有的字段来说,是带类名的每个字段的类序对 $\overline{C}\ \overline{f}$ 系列。在类 C 中方法 m 的类型,记为 *mtype*(m,C),是一系列参数类型 $\overline{B}$ 和单个结果类型 B 的序对,记为 $\overline{B}$→B。同样地,在类 C 中方法 m 的主体,记为 *mbody*(m,C),是一系列参数 $\overline{x}$ 和项 t 序对,记为($\overline{x}$,t)。谓词 *override*(m,D,$\overline{C}$→C_0)用来判断带参数类型 $\overline{C}$ 和结果类型 C_0 的方法 m 是否在 D 的子类中定义了。对重载来说,如果超类中也声明了相同名字的方法,那么它必须有相同的类型。

求值

这里用一个标准的值调用操作语义(参见图 19.3)。三个计算规则(字段访问、方法调用

① 在 Java 中,类 Object 含有几个方法,但在 FJ 中不考虑。

和强制转型)在 19.2 节中已经解释了。剩下的规则形式化了值调用策略。从正常求值结束产生的值是完全被求值的对象所创建形为 new C($\overline{\text{v}}$)的项。

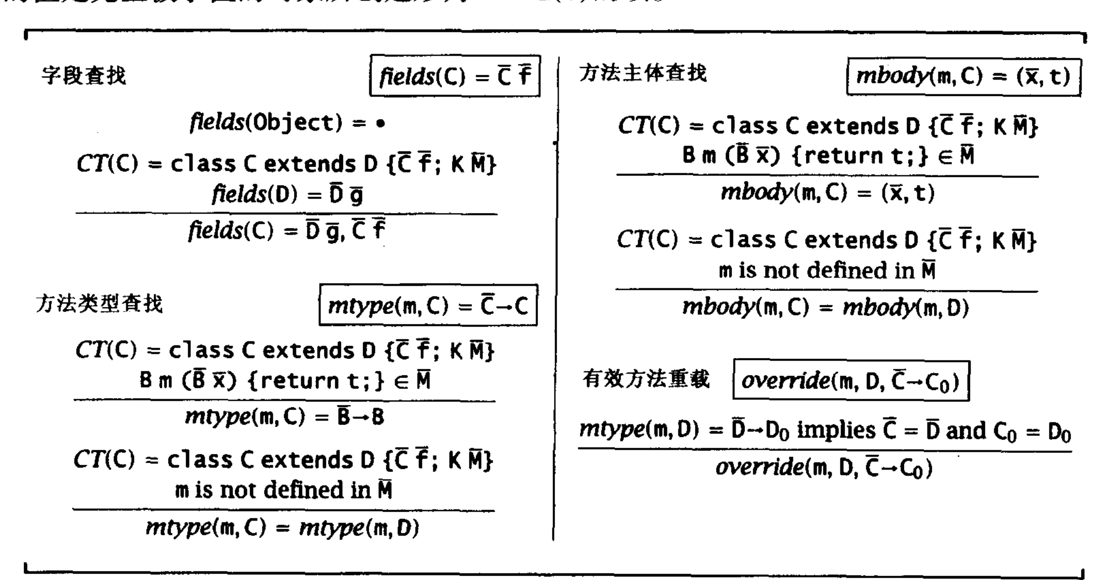


图 19.2　轻量级 Java(辅助定义)

求值　　$\boxed{\texttt{t} \longrightarrow \texttt{t}'}$

$$\frac{fields(\texttt{C}) = \overline{\texttt{C}}\;\overline{\texttt{f}}}{(\texttt{new C}(\overline{\texttt{v}})).\texttt{f}_i \longrightarrow \texttt{v}_i} \quad \text{(E-ProjNew)}$$

$$\frac{mbody(\texttt{m},\texttt{C}) = (\overline{\texttt{x}},\texttt{t}_0)}{(\texttt{new C}(\overline{\texttt{v}})).\texttt{m}(\overline{\texttt{u}}) \longrightarrow [\overline{\texttt{x}} \mapsto \overline{\texttt{u}}, \texttt{this} \mapsto \texttt{new C}(\overline{\texttt{v}})]\texttt{t}_0} \quad \text{(E-InvkNew)}$$

$$\frac{\texttt{C} <: \texttt{D}}{(\texttt{D})(\texttt{new C}(\overline{\texttt{v}})) \longrightarrow \texttt{new C}(\overline{\texttt{v}})} \quad \text{(E-CastNew)}$$

$$\frac{\texttt{t}_0 \longrightarrow \texttt{t}_0'}{\texttt{t}_0.\texttt{f} \longrightarrow \texttt{t}_0'.\texttt{f}} \quad \text{(E-Field)}$$

$$\frac{\texttt{t}_0 \longrightarrow \texttt{t}_0'}{\texttt{t}_0.\texttt{m}(\overline{\texttt{t}}) \longrightarrow \texttt{t}_0'.\texttt{m}(\overline{\texttt{t}})} \quad \text{(E-Invk-Recv)}$$

$$\frac{\texttt{t}_i \longrightarrow \texttt{t}_i'}{\texttt{v}_0.\texttt{m}(\overline{\texttt{v}}, \texttt{t}_i, \overline{\texttt{t}}) \longrightarrow \texttt{v}_0.\texttt{m}(\overline{\texttt{v}}, \texttt{t}_i', \overline{\texttt{t}})} \quad \text{(E-Invk-Arg)}$$

$$\frac{\texttt{t}_i \longrightarrow \texttt{t}_i'}{\texttt{new C}(\overline{\texttt{v}}, \texttt{t}_i, \overline{\texttt{t}}) \longrightarrow \texttt{new C}(\overline{\texttt{v}}, \texttt{t}_i', \overline{\texttt{t}})} \quad \text{(E-New-Arg)}$$

$$\frac{\texttt{t}_0 \longrightarrow \texttt{t}_0'}{(\texttt{C})\texttt{t}_0 \longrightarrow (\texttt{C})\texttt{t}_0'} \quad \text{(E-Cast)}$$

图 19.3　轻量级 Java(求值)

注意强制转型运算在运行时会检查被转型对象的实际类型是否是转型中声明类型的子类型。如果是,将不进行转型运算而返回对象本身。这正好与 Java 的语义对应:一个运行时强制转型无论如何都不会改变一个对象——它或者成功或者失败并且提升一个异常。在 FJ 中,转型失败后不会提升异常,而是被阻塞,这样求值规则根本不会用上。

类型化

图 19.4 给出了项、方法声明、类声明的类型化规则。上下文 Γ 是一个从变量到类型的有限映射,记为 $\overline{\texttt{x}}:\overline{\texttt{C}}$。项的类型化语句有形式 Γ⊢ t:C,读做"在上下文 Γ 中,项 t 有类型 C"。类型化规则是语法制导的[1],每个项有一个规则,除此之外还有三个转型规则(下面会谈到)。构

[1] 在选择类型化关系的算法规则上仍按照 Java 的方式。有趣的是,在 Java 中该选择是强制的,参见练习 19.4.6。

造子和方法调用的类型规则检查每个参数的类型是否为声明的相关形参类型的子类型。我们用明显的方式简写类型语句序列，如 $\Gamma\vdash \bar{t}:\bar{C}$ 表示 $\Gamma\vdash t_1:C_1,\cdots,\Gamma\vdash t_n:C_n$，用 $\bar{C}<:\bar{D}$ 表示 $C_1<:D_1,\cdots,C_n<:D_n$。

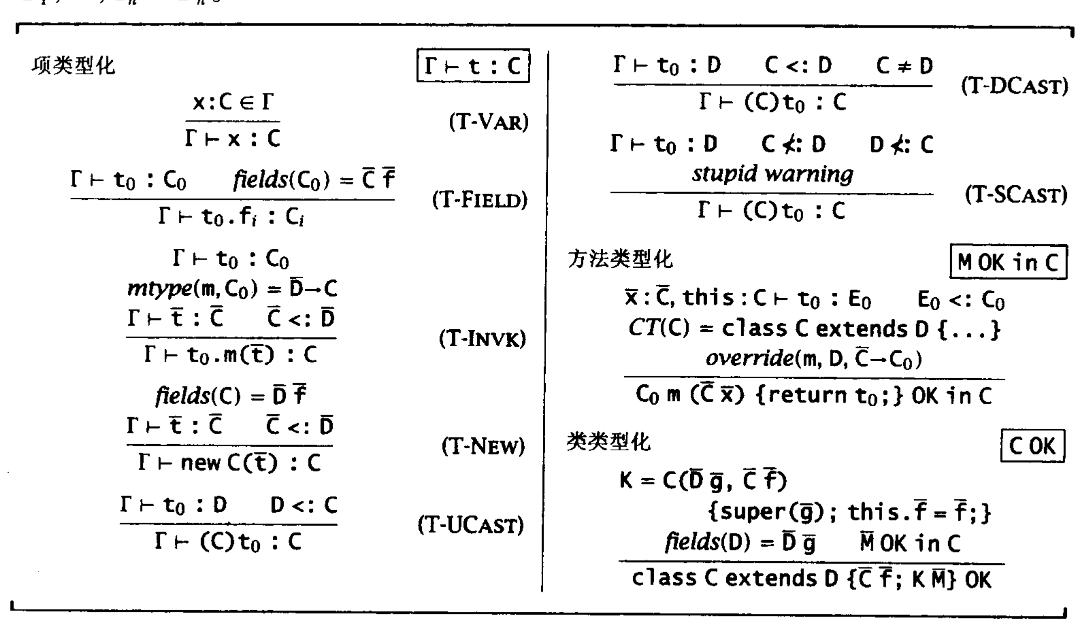


图 19.4　轻量级 Java(类型)

一个在 FJ 中较小的技术调用是引入了“愚蠢”强制转型。类型转型有三个规则：向上转型中被转型者是目标的子类，向下转型的则是目标的超类，而愚蠢转型与目标无任何关系。Java 编译器拒绝含愚蠢转型的不良类型项，但是如果要将类型安全性作为一小步语义的类型保持定理而形式化，我们必须允许在 FJ 中存在愚蠢转型。因为一个明智的项可能归约为一个包含愚蠢转型的项。例如，用 19.2 节中定义的类 A 和类 B，考虑下面的例子：

```
(A)(Object)new B() ⟶ (A)new B()
```

在愚蠢转型的类型规则(T-SCast)中包括了假设 stupid warning(愚蠢警告)，以此来说明愚蠢转型的特殊性质；如果一个 FJ 类型不包含这个规则，则对应一个合法的 Java 类型。

方法声明的类型语句形式为 M OK in C，读做“如果方法声明 M 发生在类 C 中，则是良形式的”。它在方法体中用了项类型化关系，其中自由变量是有声明类型的方法参数，以及还有类型为 C 的特殊变量 this。

类声明的类型语句有形式 CL OK，读做“类声明 CL 是良形式的”，它检查构造子应用 super 到超类的字段中并初始化该类中声明的字段，还要检查类中每个方法声明都正确。

项的类型可能依靠于它调用的方法类型，而方法类型依靠于方法体中项的类型，因此有必要检查它是否不存在不良定义的循环。确实不存在：因为每个方法的类型都是明确定义了的，所以这种循环被打破。加载类表并在类表中的类被检查完之前用类表来做类型检查是有可能的，只要每个类最终被检查。

19.4.2　练习[★★]：在 FJ 中，许多设计理念都受到希望它成为 Java 的子集想法的影响，这样每个良的或不良的类型化 FJ 程序都可作为一个良的或不良的类型化 Java 程序，且良类

型的程序行为都一样。假设不考虑这个要求，即假设所要的是类 Java 的核心演算。你将怎样改变 FJ 的设计来使它更简单或者更精制？

19.4.3　练习[推荐，★★★ ↛]：为了简化表示，FJ 中忽略了将新值赋给对象的某字段的操作，但可以把它加进来，而不会很大地影响演算的基本特征。仿照第 13 章对引用的处理方式来试着加加看。

19.4.4　练习[★★★ ↛]：仿照第14章对异常的处理方式，用类似 Java 中的 raise 和 try 形式来扩展 FJ。

19.4.5　练习[★★ ↛]：如全 Java 一样，FJ 以算法的形式来表示类型化关系。没有包含规则；但几个其他规则在它们的前提中包括子类型化检查。如果删除大部分或全部符合单个包含规则的这些前提，系统能以一个更易的表述方式重新形式化吗？

19.4.6　练习[★★★]：全 Java 提供类和接口，具体指出方法类型，而不是它们的实现方式。接口是有用的，因为它们允许一个更丰富，非树形结构的子类型关系：每个类有一个单独的超类（从那里继承实例变量和方法体），但是又可能实现多次接口。
在 Java 中，接口的存在实际上强制使用类型化关系的算法表示，它给每个可类型化的项惟一的（最小的）类型。原因是在条件表达式（在 Java 中记为 $t_1?t_2:t_3$）和接口之间会产生互相影响。

1. 显示怎样以 Java 的风格扩展带有接口的 FJ。
2. 请说明存在接口时，子类型关系在合类型下没必要封闭（回忆 16.3 节中合类型在条件表达式的最小类型化特性中起关键性作用）。
3. Java 中条件表达式的类型化规则是什么？它合理吗？

19.4.7　练习[★★★]：FJ 包括 Java 的 this 关键字，但是省略了 super。说明一下怎样把它加上。

19.5　性质

这里将证明一个标准的 FJ 类型保持定理。

19.5.1　定理[保持]：如果 $\Gamma\vdash t:C$ 并且 $t\to t'$，那么对一些 C'<:C 有 $\Gamma\vdash t':C'$。

证明：作为练习[★★★]。

我们也能说明标准进展定理的一个变化：如果一个程序是良类型的，那么它发生阻塞的惟一可能是在它无法实现一个向下转型时。在出现后者情况下，可用求值上下文机制来找出失败的向下转型。

19.5.2　引理：假设 t 是一个良类型项：

1. 如果 $t=\text{new}C_0(\bar{t}).f$，那么有 $fields(C_0)=$ 和 $\bar{C}\ \bar{f}$ 和 $f\in\bar{f}$。
2. 如果 $t=\text{new}C_0(\bar{t}).m(\bar{s})$，那么有 $mbody(m,C_0)=(\bar{x},t_0)$ 和 $|\bar{x}|=|\bar{s}|$。

证明：直接可证明。

19.5.3　定义：FJ 中的求值上下文定义如下：

E ::=		求值上下文：
	[]	括号
	E.f	字段访问
	E.m($\overline{t}$)	方法调用(接收方)
	v.m($\overline{v}$,E,$\overline{t}$)	方法调用(参数)
	new C($\overline{v}$,E,$\overline{t}$)	对象创建(参数)
	(C)E	强制转型

每个求值上下文是带括号的项(写为[])。我们用 E[t]表示用 t 代替 E 中的括号而得到的一般项。

求值上下文抓住了“将被归约的下一个子项”这个概念，在一定意义上，如果，根据 E-ProjNew，E-InvkNew 和 E-CastNew 中的一个计算规则，对于惟一的 E，r 和 r′，且 r→r′，可将 t 和 t′表示为 t = E[r]和 t′ = E[r′]。

19.5.4　定理[进展]：假设 t 是一个封闭的、良类型的范式，那么或者(1)t 是一个值，或者(2)对一些求值上下文 E，可将 t 表示为 t = E[(C)(new D($\overline{v}$))]，其中 D $\not<:$ C。

证明：直接对类型推导进行归纳。

进展特性可被再增强一点：如果 t 仅仅包含向上转型，那么它不能被阻塞(并且，如果原来的程序仅包括向上转型，那么求值将永远不会产生任何非向上的强制转型)。但是，一般情况下，我们想通过强制转型来降低对象的静态类型，所以运行时有可能发生强制转型失败。在全 Java 中，当然，转型失败不会停止整个程序：它会产生一个异常，该异常能被附近的异常处理器捕获。

19.5.5　练习[★★★ ↛]：从一个 λ 演算的类型检查器开始，为轻量级 Java 建立一个类型检查器和解释器。

19.5.6　练习[★★★★ ↛]：原始 FJ 论文(Igarashi，Pierce 和 Wadler，1999)也形式化了 GJ 风格中的多态类型。将这些特征扩展到练习 19.5.5 的类型检查器和解释器中(在做本练习之前，可能需要先阅读一下第 23 章、第 24 章、第 25 章和第 28 章)。

19.6　编码及初始对象

我们已经看到语义和简单面向对象语言类型化的两个对比的方法。在第 18 章中，我们用简单类型 λ 演算的特征与记录、引用和子类型化等的组合形式来编码对象、类和继承。在本章中，我们给出一个简单语言(以对象和类为初始机制)的直接描述。

每个方法都有它的用处。研究对象编码应放在基本的封装和重用机制，并且允许与其他相同目的的机制做比较。这些编码还有助于理解对象被编译器翻译为更低级语言的方式，也有助于理解对象和其他语言特征之间的联系。另一方面，将对象看成原语可帮助我们直接讨论它们的操作语义和类型化功能；这种表示方式是高级语言设计和文档编制的较好工具。

最后，我们还应该持以下观点：一个带有自己的类型化规则和操作语义的高级语言(包括原始特性，如对象和类等)，拥有一个从该语言到一些含记录和函数的低级语言(甚至是一个只有注册器、指针和指令序列的更低级语言)的翻译程序，以及一个证明该翻译是高级语言中正确实现的程序证明程序，即能证明翻译保持了高级语言的求值和类型化特征的证明程序。关

于这点有多种观点——对 FJ,参见 League,Trifonov 和 Shao(2001)等人的著作,对其他面向对象核心演算,参见 Hofmann 和 Pierce(1995b),Bruce(1994,2002),Abadi,Cardelli 和 Viswanathan(1996)等人的著作。

19.7 注释

这一章是根据 Igarashi,Pierce 和 Wadler(1999)写的 FJ 文章改编而成。在表示方式上,主要区别是,这里使用的是一种值调用操作语义,而最初用的是非确定性 β 归约关系,这样做是为了与后面的章节保持一致。

对于 Java 子集类型的安全问题有几种证明方法。最早是 Drossopoulou,Eisenbach 和 Khrshid(1999)等,用一个技巧[后来经过了 Syme(1997)系统的检验],证明了系列 Java 真子集的安全性。像 FJ,用一小步操作语义,但通过完全忽略强制转型而避免了出现“愚蠢转型”。Nipkow 和 Oheimb(1998)给出一个大一些的核心语言的系统安全性证明。其语言包括强制转型,但是它通过用一大步操作语义形式化,这样回避了愚蠢转型问题。Flatt,Krishnamurthi 和 Felleisen(1998a,1998b)用一小步语义和形式化了带赋值和转型的语言,和 FJ 中一样处理愚蠢转型。他们的系统是比 FJ 大一些(语法、类型和操作语义规则占三倍空间),它的安全性证明,虽然较长,但复杂度相似。

在上面这三个研究中,Flatt,Krishnamurthi 和 Felleisen 最接近 FJ 的重要原因是:他们的目标和这里一样,是选择一个尽可能小的核心演算,采用的是与完成某些特殊任务有关的 Java 特征的想法。他们提出的任务是分析一个带有公用 LISP 风格的 mixins 的 Java 扩展。在上面提到的另外两个系统的目标包括尽可能地和 Java 的子集一样大,虽然他们最初的兴趣是证明 Java 本身的安全性。

有关面向对象语言基础的著作还包括很多形式化基于类的面向对象语言方面的文章,内容或者涉及将类作为原语(如 Wand 1989a,Bruce 1994,Bono Patel Shmatikov 和 Mitchell 1999b,Bono Patel 和Shmatikov 1999a)或者把类翻译为低级机制(例如,Fisher 和 Mitchell 1998,Bono 和 Fisher 1998,Abadi 和Cardelli 1996,Pierce 和 Turner 1994)。

有关的工作还涉及面向对象语言模型方面,该模型中类是由方法重载或委托的形式所取代的(Ungar 和 Smith,1987),这样单个对象可以继承其他对象的行为。结果演算会比那些基于类的语言简单,因为它们处理的是更小的概念集。关于这方面最深入及最权威的是 Abadi 和 Cardelli 的对象演算(1996)。另一个比较流行的是由 Fisher,Honsell 和 Mitchell(1994)等人提出的。

一个关于对象、类和继承等方面有点不同的方法——多方法,已由 Castagna,Ghelli 和 Longo(1995)进行了形式化。在 18.1 节的脚注中给出了说明。

每个大型语言体内是挣扎要走出来的小的语言……

——Igarashi,Pierce 和 Wadler(1999)

每个大的程序中都有一个小程序挣扎着走出来……

——Tony Hoare《大程序有效介绍》(1970)

我很胖,但我里面很瘦。

如果你发现每个胖子体内躲着一个瘦子,你会感到震撼吗?

——George Orwell,Coming Up For Air(1939)

第四部分　递 归 类 型

第 20 章　递归类型简介[1]

在 11.12 节谈到了怎样将一个简单类型系统扩展成包括类型构造子 List(T),其中的元素是类型 T 的元素列表。列表仅仅是共同结构(包括队列、二叉树、标签树、抽象语义树等)的大类中的一个例子,可能任意长短,但又结构简单、有规则。例如,List(Nat)的元素可为 nil,也可为一个数字及另一个 List(Nat)构成的序对(即“cons 单元”)。显然,如果将这些结构中的每一个都作为独立的、原始的语言特征是没有意义的。相反,我们需要一个总的机制,通过它将简单元素定义为这些结构。这种机制被称为递归类型。

再考虑数字列表的类型[2]。用 11.10 节和 11.7 节中定义的变化类型和元组类型,这里列表的表示可为 nil 或一个序对:

```
NatList = <nil:Unit, cons:{...,...}>;
```

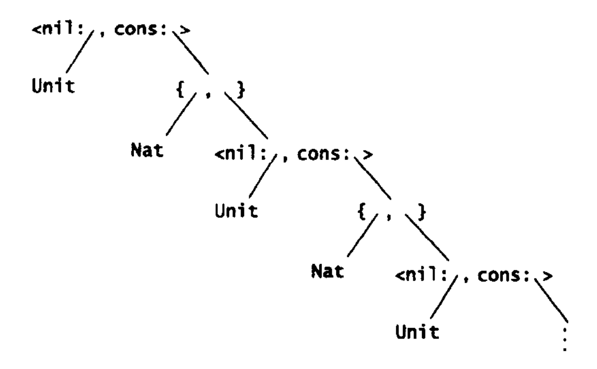
nil 表示平凡数据值,因为 nil 标记本身已经告诉我们列表为空。而 cons 标记所带的值是包含数字和另一个列表的序对。序对的第一个分量的类型为 Nat:

```
NatList = <nil:Unit, cons: {Nat, ...}>;
```

而第二个分量是一列数字,即所定义的 NatList 型元素:

```
NatList = <nil:Unit, cons:{Nat,NatList}>;
```

该等式不是一个简单的定义(也就是说,这里不是给已经明白意思的表达式一个新的名称),因为右端提到的名称正是我们要定义的。这里可将它看做一个无穷树:



可以说明该无穷树类型的递归等式与 5.1 节说明的递归 factorial 函数有些类似。和那里的方式一样,这里可方便地将“循环”移到等号(=)的右端使之定义更正确[3]。这里对类型引入一个明确的递归操作符 μ:

```
NatList = μX. <nil:Unit, cons:{Nat,X}>;
```

---

[1] 本章学习的系统为带递归类型的简单类型演算。其中的例子使用了前几章介绍过的特征;相关的检查器为 fullequirec。对 20.2 节,相关的检查器为 fullisorec。

[2] 本章的后面将不考虑如何用任意类型 T 的元素来定义单个的、一般化的列表。为解决这个问题,需要引入类型操作符的概念,关于这一点将在第 29 章中介绍。

[3] 说它方便的原因是可以在讨论递归类型时不需要给它们名称。然而,给出明确名称的递归类型有时处理起来也很方便,请参阅 19.3 节的规范化和结构化类型系统中的说明。

直观上,这个定义读做“将 NatList 定义为满足 X = <nil:Unit,cons:{Nat,X}> 的无穷的类型”。

在 20.2 节中,将会看到实际上有两种不同形式化递归类型的方式(分别称为相等递归和同构递归),不同之处是程序员以类型注释的方式对类型检查器的帮助程度。在接下来的程序实例中,我们使用简化的相等递归表示方式。

20.1 实例

列表

首先,结束上面开始的数字列表例子。为了用列表编程,需要一个常量 nil,一个在列表头添加元素的构造子 cons,一个取得列表并且返回一个布尔值的 isnil 操作,删除非空列表的列表头和列表尾的析构函数 hd 和 tl。和图 11.13 中的内置操作的定义方式一样,来定义这里的操作;这里将从更简单的部分开始定义。

直接根据 NatList 的定义将 nil 和 cons 定义为两字段的变式类型:

```
nil = <nil=unit> as NatList;
▸ nil : NatList

cons = λn:Nat. λl:NatList. <cons={n,l}> as NatList;
▸ cons : Nat → NatList → NatList
```

(回想第 11.10 节中形为 <l=t> as T 的表达式是用来引入变式类型的值:值 t 被标记为 l,并且被“注入”变式类型 T 中。另外,注意这里的类型检查器是自动地将递归类型 NatList“展开”为变式类型 <nil:Unit,cons:{Nat,NatList}>)。

关于列表的基本操作还包括检查它们的结构和截取适当的部分。它们都在case语句中完成:

```
isnil = λl:NatList. case l of
                      <nil=u> ⇒ true
                    | <cons=p> ⇒ false;
▸ isnil : NatList → Bool

hd = λl:NatList. case l of <nil=u> ⇒ 0 | <cons=p> ⇒ p.1;
▸ hd : NatList → Nat

tl = λl:NatList. case l of <nil=u> ⇒ l | <cons=p> ⇒ p.2;
▸ tl : NatList → NatList
```

我们任意地定义一个空列表的hd为0,空列表的tl为空列表。但在这种情况下可能会出现异常。

有了这些定义,可以用它们写一个求和的递归函数:

```
sumlist = fix (λs:NatList→Nat. λl:NatList.
                if isnil l then 0 else plus (hd l) (s (tl l)));
▸ sumlist : NatList → Nat

mylist = cons 2 (cons 3 (cons 5 nil));
sumlist mylist;
▸ 10 : Nat
```

注意,尽管 NatList 本身是一个无穷长的类型表达式,但它的所有元素是有限的列表(因为没有办法用序对和标记原语或值调用 fix 来构造无穷大的结构)。

20.1.1 练习[★★]:带标签的二叉树将树结构定义为一个树叶(不带标签)或者带数字标签和两个子树的内部节点。定义一个类型 NatTree 及其上合适的一些操作,如构造、析构、测试树,等等。写一个深度优先遍历的函数,使之返回找到的标签列表。用 fullequirec 检查器来检查你的代码。

饥饿函数

这里有个稍微复杂一点的递归类型的例子,称为饥饿函数,它可接受任何以数字为参数的数值,并生成一个需要更多数值的饥饿函数:

```
Hungry = μA. Nat→A;
```

这种类型的元素能用 fix 算子来定义:

```
  f = fix (λf: Nat→Hungry. λn:Nat. f);
▸ f : Hungry

  f 0 1 2 3 4 5;
▸ <fun> : Hungry
```

流

对上面饥饿类型的一个更有用的变式是 Stream 类型,它能用 unit 类型的任意值,每次返回一个数字和新的流的序对:

```
Stream = μA. Unit→{Nat,A};
```

我们能为流定义两个"析构函数";如果 s 是一个流,那么 hd s 是传递给它 unit 时返回的第一个数字:

```
  hd = λs:Stream. (s unit).1;
▸ hd : Stream → Nat
```

同样地,tl s 是传递 unit 到 s 时,获得的新的流:

```
  tl = λs:Stream. (s unit).2;
▸ tl : Stream → Stream
```

为了构造一个流,和上面一样用 fix:

```
  upfrom0 = fix (λf: Nat→Stream. λn:Nat. λ_:Unit. {n,f (succ n)}) 0;
▸ upfrom0 : Stream

  hd upfrom0;
▸ 0 : Nat

  hd (tl (tl (tl upfrom0)));
▸ 3 : Nat
```

20.1.2 练习[推荐,★★]:定义一个产生连续的Fibonacci序列元素流(1,1,2,3,5,8,13,…)。

流能进一步归纳为一个简单形式 process，它是接受一个数字并返回一个数字和一个新进程的函数：

```
Process = μA. Nat→{Nat,A};
```

例如，这里有一个进程，在每一步会返回已给出的所有数字的和：

```
p = fix (λf: Nat→Process. λacc:Nat. λn:Nat.
            let newacc = plus acc n in
            {newacc, f newacc}) 0;
▸ p : Process
```

正如为流做的，这里定义一个作用在进程上的辅助函数：

```
 curr = λs:Process. (s 0).1;
▸ curr : Process → Nat

 send = λn:Nat. λs:Process. (s n).2;
▸ send : Nat → Process → Process
```

如果赋给进程 p 数值 5,3 和 20，那最后一步返回的数字是 28：

```
 curr (send 20 (send 3 (send 5 p)));
▸ 28 : Nat
```

对象

若对上面的例子稍做调整可得到另一个熟悉的数据关系：对象。例如，这是计数器对象，它记录一个数字，并且允许我们查询或者增加它：

```
Counter = μC. {get:Nat, inc:Unit→C};
```

注意，这里处理对象的方式是纯函数式的（类似于第 19 章中，但与第 18 章中的不同）：给计数器对象发送 inc 消息不能引起这个对象改变它的内部状态；而是，操作将返回一个内部状态增加后的新的计数器对象。递归类型的使用使得返回的对象和原来的对象类型完全相同。

在这些对象和上面讨论的进程间惟一的不同是对象是递归定义的记录（包含一个函数），而进程是一个递归定义函数（返回一个元组）。考虑到这个变化是有用的原因，我们能扩展记录使之包含多个函数，例如，一个减少操作：

```
Counter = μC. {get:Nat, inc:Unit→C, dec:Unit→C};
```

为了创建一个计数器对象，如上所做，这里使用不动点组合算子：

```
 c = let create = fix (λf: {x:Nat}→Counter. λs: {x:Nat}.
                         {get = s.x,
                          inc = λ_:Unit. f {x=succ(s.x)},
                          dec = λ_:Unit. f {x=pred(s.x)} })
     in create {x=0};

▸ c : Counter
```

为了利用 c 的一个操作，我们简单地投影出适合的字段：

```
 c1 = c.inc unit;
 c2 = c1.inc unit;
 c2.get;

▸ 2 : Nat
```

20.1.3 练习[推荐,★★]:扩展 Counter 类型和上面的计数器 c,增加 backup 和 reset 操作(如在 18.7 节中做的):调用 backup 引起计数器在一个独立的内部寄存器存储它的当前值;调用 reset 使计数器的值由寄存器的值重新设置。

递归类型的递归值

一个更加令人吃惊的递归类型的用法(更明显体现了它们的表达能力)是不动点组合算子的良类型实现方式。对 T 的任何类型,可为关于 T 的函数定义如下一个不动点构造子:

```
fixT = λf:T→T. (λx:(μA.A→T). f (x x)) (λx:(μA.A→T). f (x x));
▸ fixT : (T→T) → T
```

注意,如果我们抹除类型,该项就是 5.2 节递归类型里看到的无类型不动点组合算子。

这里关键的技巧是用一个递归类型来类型化子表达式 x x。正如在练习 9.3.2 中看到的,类型化该项需要 x 有一个箭头类型,它的定义域是类型 x 本身。显然,这个特性没有有限的类型,而无穷类型 $\mu A.A\to T$ 完美地完成这个工作。

这个例子的推论是递归类型的存在打破了强规范化特性:我们能用 fix_T 组合算子来写一个良类型的项使它的求值(当应用于 unit 时)发散:

```
divergeT = λ_:Unit. fixT (λx:T. x);
▸ divergeT : Unit → T
```

而且,因为能获得该项的每种类型的形式,所以该系统中每种类型都能用(不像 $\lambda_{\to}$)[①]。

无类型 λ 演算,约式

可能对递归类型最有力的说明是将整个无类型 λ 演算(以一种良类型的方式)嵌入含递归类型的静态类型化语言中。让 D 成为如下的类型[②]:

```
D = μX.X→X;
```

定义一个将 D 到 D 的函数映射为 D 的元素的"投射函数"lam:

```
lam = λf:D→D. f as D;
▸ lam : D
```

为了将 D 的一个元素应用到另一个,我们简单展开第一个类型,产生一个函数,又将该函数应用到第二个:

```
ap = λf:D. λa:D. f a;
▸ ap : D
```

现在,假设 M 是一个只含变量、抽象和应用的封闭的 λ 项。那么能构造一个表示 M 的 D 的元素,记为 M^*(如下统一的形式):

---

① 这一点使带递归类型的系统从逻辑上来说是无用的:根据 Curry-Howard 对应(参见 9.4 节),如果将类型解释为逻辑命题,并将"类型 T 是适用的"理解为"命题 T 是可证明的",那么每个类型都可适用的意思就变成每个命题在逻辑上都是可证明的,即逻辑是不一致的。

② 熟悉外延语义的读者会注意到 D 的定义其实是纯 λ 演算的语义模型中使用的全域性质的定义。

```
x*        =  x
(λx.M)*   =  lam (λx:D. M*)
(M N)*    =  ap M* N*
```

例如,这是表示为 D 的一个元素的无类型不动点组合算子:

```
fixD = lam (λf:D.
           ap (lam (λx:D. ap f (ap x x)))
              (lam (λx:D. ap f (ap x x))));

▸ fixD : D
```

纯 λ 演算的嵌入可扩展为含一些特征(如数字)。我们将 D 的定义改为一个带标记的数字变式类型及一个带标记的函数变式类型:

```
D = μX. <nat:Nat, fn:X→X>;
```

也就是说,D 的元素或是数字,或者是从 D 到 D 的函数,分别标记为 nat 和 fn。lam 构造子的实现,基本与以前一样:

```
lam = λf:D→D. <fn=f> as D;

▸ lam : (D→D) → D
```

然而,ap 以不同的方式实现(十分有趣):

```
ap = λf:D. λa:D.
       case f of
         <nat=n> ⇒ divergeD unit
       | <fn=f> ⇒ f a;

▸ ap : D → D → D
```

将 f 应用到 a 之前,需要用一个 case 语句从 f 中提取一个函数。这迫使我们说明当 f 不是函数时应用该如何执行(这个例子仅发散;但也能提升异常)。注意,这里的标记检查与动态类型化语言如 Scheme 执行时的标记检查多么相似呀!从这种意义上,类型化计算可认为是"包括"无类型的或动态的类型计算。

为给 D 的元素定义后继函数,仍需要类似的标记检查:

```
suc = λf:D. case f of
              <nat=n> ⇒ (<nat=succ n> as D)
            | <fn=f> ⇒ divergeD unit;

▸ suc : D → D
```

D 中添入 0,结果为:

```
zro = <nat=0> as D;

▸ zro : D
```

20.1.4 练习[★]:用布尔型和条件式来扩展该代码,将项 if false then 1 else 0 和 if false then 1 else false 作为 D 的元素编码。当求值这些项时,会发生什么?

20.1.5 练习[推荐,★★]:扩展数据类型 D 使之包括记录:

```
D = μX. <nat:Nat, fn:X→X, rcd:Nat→X>;
```

并实现记录构造和字段投影。为简单化起见，用自然数字作为字段标签，即记录可表示为从自然数到 D 元素的函数。可用 fullequirec 检查器来测试你的扩展。

20.2 形式

在关于类型系统的著作里，存在两个基本的递归类型方法。它们本质的差别就在它们对一个简单问题的回答上：类型 μX.T 和它的一步展开间的关系是什么？例如，在 NatList 和 < nil: Unit, cons: {Nat, NatList} > 间的关系是怎样的？

1. 相等递归(equi-recursive)方法是把这两个类型表达式作为相同定义(在所有上下文可交换)，因为它们代表同样的无穷树[1]。类型检查器有责任确保某类型的项可以作为一个参数传给一个希望得到另一种类型参数的函数。
   关于相等递归方法令人高兴的事是：它使类型表达式可以为无穷[2]，这一方式是对我们已理解的系统声明方式的惟一改变。只要已有的定义、安全性定理及证明不是依靠对类型表达式(不再起作用)进行归纳，则都不会发生改变。
   当然，相等递归类型的实现还需要一些工作，因为类型检查算法不能直接作用于无穷结构。这一点如何做到将是第 21 章讨论的话题。
2. 另一方面，同构递归(iso-recursive)将一个递归类型与其展开式视为不同(但同构)。
   形式上，递归类型 μX.T 的展开式是将主体 T 取出并将出现 X 的地方用整个递归类型来代替，即使用标准的代换符号：$[X \mapsto (\mu X.T)]T$。例如，类型 NatList 即为：

```
μX.<nil:Unit,cons:{Nat,X}>
```

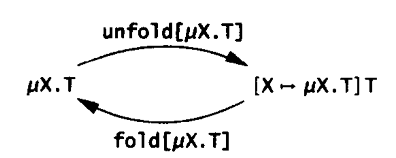
   展开为：

```
<nil:Unit, cons:{Nat, μX.<nil:Unit,cons:{Nat,X}>}>
```

   在同构递归类型的系统里，为每个递归类型 μX.T 引入了如下一对函数：

```
unfold[μX.T]   :  μX.T → [X ↦ μX.T]T
fold[μX.T]     :  [X ↦ μX.T]T → μX.T
```

   如下图通过两个类型之间相互映射可“目击同构的过程”：



   fold 和 unfold 映射已作为语言的原语形式，正如在图 20.1 所描述的。求值规则 E-UnfldFld 说明了它们是同构的，因为根据该规则，当 fold 遇到了对应的 unfold 时会被消去(求值规则不要求 fold 和 unfold 的注释是相同的，因为在运行时必须用类型检查器来检查该条件是否满足。然而，在良类型程序的求值中，无论何时运用 E-UnfldFld，这两种类型注释都相等)。

---

① 从 μ 类型到它们的无穷树的映射，将在 21.8 节中有精确的定义。

② 严格地说，应称为有规则的，参见 21.7 节。

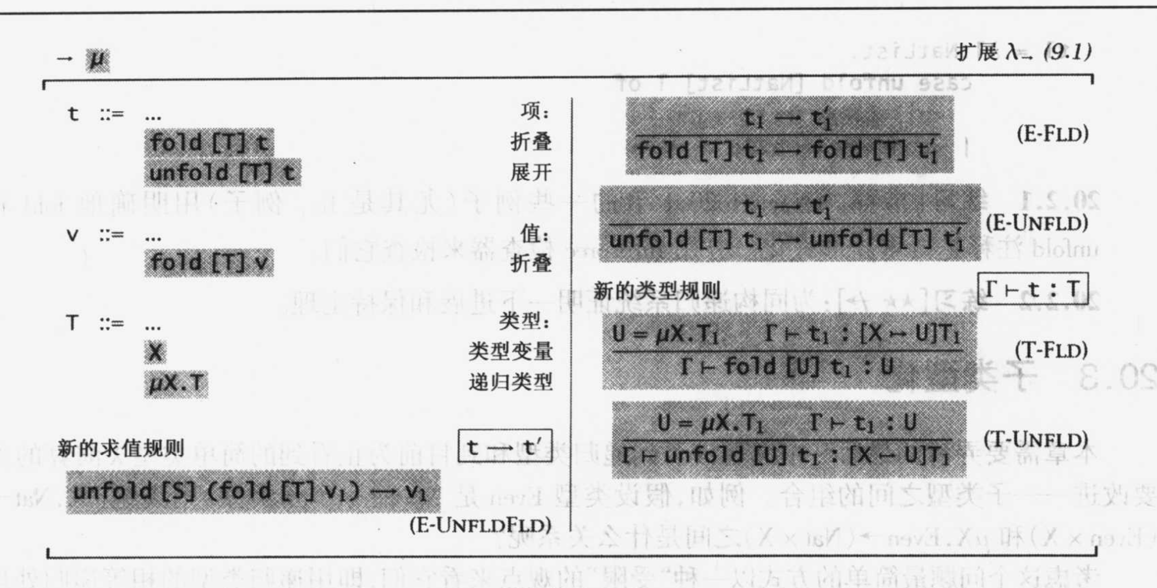


图 20.1　类似递归类型(λμ)

这两种方法广泛地运用于理论学习和程序语言设计中。相等递归风格可说更直观些,但对类型检查器的要求更高,因为它必须有效地推断出 fold 和 unfold 注释的出现位置。此外,相等递归类型与其他高级类型特征,如囿量词和类型操作符之间的相互作用相当复杂,会导致严重的理论困难(如 Ghelli,1993;Colazzo 和 Ghelli,1999),甚至是不可决定的类型检查问题(Solomon,1978)。

同构递归风格在表示上有些累赘,在使用递归类型的地方,它要求程序要带上 fold 和 unfold 说明。然而,这些注释会与其他注释结合而被"隐藏"。例如,在 ML 系列语言中,每个 datatype 定义都暗含了递归类型。每用一个构造子来建立一个数据类型的值时都隐含地使用了 fold,而每个出现在模式匹配的构造子都会隐含地用到 unfold。相似地,在 Java 中每个类定义隐含地引入了递归类型,且调用对象中的一个方法就涉及到了unfold。这种合适的机制重叠使同构递归风格在实践中相当受欢迎。

例如,这里是一个同构递归形式的 NatList 例子。首先,为 NatLiat 的展开形式定义一个缩写形式是很方便的:

```
NLBody = <nil:Unit, cons:{Nat,NatList}>;
```

现在,通过建立类型为 NLBody 的变式类型来定义一个 nil,然后把它折叠为 NatList;对 cons 也这样做:

```
nil = fold [NatList] (<nil=unit> as NLBody);
cons = λn:Nat. λl:NatList. fold [NatList] <cons={n,l}> as NLBody;
```

相反,isnil,hd 和 tl 操作的定义需要将 NatList 视为一个变式类型,以便它们能对它的标记进行分析。需要将参数 l 展开来实现:

```
isnil = λl:NatList.
            case unfold [NatList] l of
               <nil=u> ⇒ true
             | <cons=p> ⇒ false;
hd = λl:NatList.
         case unfold [NatList] l of
            <nil=u> ⇒ 0
          | <cons=p> ⇒ p.1;
```

```
tl = λl:NatList.
       case unfold [NatList] l of
         <nil=u> ⇒ l
       | <cons=p> ⇒ p.2;
```

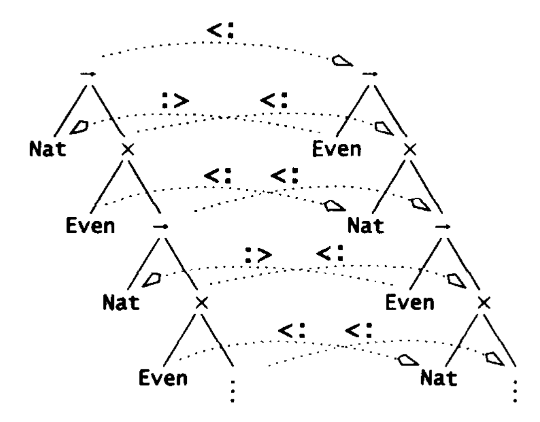
20.2.1 练习[推荐,★★]: 对 20.1 节的一些例子(尤其是 fix_T 例子)用明确地 fold 和 unfold 注释进行重新形式化。并用 fullisorec 检查器来检查它们。

20.2.2 练习[★★ ↛]: 为同构递归系统证明一下进展和保持定理。

20.3 子类型化

本章需要弄清的最后一个问题涉及到递归类型和到目前为止看到的简单类型 λ 演算的主要改进——子类型之间的组合。例如,假设类型 Even 是 Nat 的一个子类型,则类型 $\mu X.\text{Nat}\to(\text{Even}\times X)$ 和 $\mu X.\text{Even}\to(\text{Nat}\times X)$ 之间是什么关系呢?

考虑这个问题最简单的方式以一种"受限"的观点来看它们,即用递归类型的相等递归处理方式。在目前的例子中,出现在两种类型中的元素可被认为是简单的反应进程(参见 20.1 节):给出一个数,它们返回另一个数及一个新的准备接收数字的进程,等等。属于第一种类型的进程总是产生偶数并且能接受任意数字。属于第二种类型的进程会产生任意数字,但总是希望赋给偶数。对函数必须接受什么参数及必须返回什么结果的限制对第一种类型要求更严格些,所以我们希望第一种类型为第二种的子类型。下面将演算以图的形式总结一下:



这种直观上的参数能精确吗? 事实上它能,我们将在第 21 章看到。

20.4 注释

在计算科学中递归类型至少回溯到 Morris(1968)写的论著。基本的语法和语义特性(不含子类型)收集在 Cardone 和 Coppo(1991)的论著中。Courcelle(1983)等研究了无穷和规则树的特性。Huet(1976)和 MacQueen, Plotkin 和 Sethi(1986)早期的论文提出了不含子类型递归类型的基本语法和语义特性。而 Abadi 和 Fiore(1996)等对同构和相等递归类型系统进行过研究。在 21.12 节中还能找到对含子类型的递归类型的其他引用。

Morris(1968,参见 pp.122-124)首次提到递归类型能用来为项构造一个良类型 fix 操作(参见 20.1 节)。

自从最早对递归类型投入工作以来,递归类型的两种形式已经发展得十分成熟了,但是最近才由 Crary, Harper 和 Puri(1999)给它们取了同构递归和相等递归这两个好记的名称。

第21章 递归类型元理论[①]

在第20章中，我们看到两种递归类型的表示形式：与其展开式定义上等价的相等递归类型和等价性是可从 fold 和 unfold 项明显看出来的同构递归类型。在本章中，我们将深入讨论相等递归类型的类型检查器的理论基础（相比之下，同构递归类型要简单些）。这里要处理的是一个包含递归类型和子类型的系统，因为实际中两者经常结合使用。一个含相等递归类型但不含子类型的系统会简单一些，因为这样只需要检查递归类型的等价性。

在第20章中，我们得知通过无穷类型上的无穷子类型推导可以帮助直观地理解相等递归类型的子类型化。这里要做的就是用共归纳的数学框架来精确地描述这种想法，并用实际的子类型算法在无穷树和推导与有限表达方式之间绘制精确联系。

从21.1节开始先回顾一下关于归纳和共归纳定义的基本理论，以及与它们相关原理的证明。21.2节和21.3节将考虑子类型化的情况对该一般理论进行实例化，定义所熟悉的有限类型上的归纳性子类型关系和它在无穷类型上的共归纳性扩展。21.4节将简单提一下与传递性规则有关的一些问题（如我们所知，它在子类型系统中是一个著名的难题）。21.5节将得出在归纳性和共归纳性定义的集合中检查成员关系的简单算法；而21.6节中将得出更精确的算法。这些算法将应用于21.7节中提到的一些重要的特殊情况——“有规则的”无穷类型的子类型。21.8节将介绍一种可以代表无穷类型的有限符号 μ 类型，然后证明 μ 类型上更为复杂的（但为有限可执行的）子类型关系符合无穷类型之间的子类关系的通常共归纳定义。21.9节将证明 μ 类型的子类型算法可终止性。21.10节将该算法与 Amadio 和 Cardelli 提出的算法进行比较。最后21.11节简单讨论一下同构递归类型。

21.1 归纳和共归纳

假设这里用全集 $\mathcal{U}$ 来作为讨论归纳定义和共归纳定义的范围。$\mathcal{U}$ 代表“世界上的任何事物”，而归纳定义或共归纳定义起到的作用就是挑选出 $\mathcal{U}$ 的子集（接下来，让 $\mathcal{U}$ 成为所有类型序对的集合，这样 $\mathcal{U}$ 的子集也是类型上的关系。在现有的讨论中，任意集合 $\mathcal{U}$ 都满足这个要求）。

21.1.1 定义： 如果 $X \subseteq Y$，则有 $F(X) \subseteq F(Y)$，那么函数 $F \in \mathcal{P}(\mathcal{U}) \to \mathcal{P}(\mathcal{U})$ 就是单调的［其中 $\mathcal{P}(\mathcal{U})$ 是 $\mathcal{U}$ 所有子集的集合］。

接下来假设 F 是 $\mathcal{P}(\mathcal{U})$ 上的某个单调函数。通常把 F 作为一个产生函数。

21.1.2 定义： 使 X 作为的 $\mathcal{U}$ 一个子集：

1. 如果 $F(X) \subseteq X$，那么 X 是 F 封闭的。

---

① 本章研究的系统是含子类型（参见图15.1）、乘积（参见图11.5）和相等递归类型的简单类型演算。相应的检查器为 equirec。

2. 如果 $X \subseteq F(X)$，那么 X 是 F 一致的。
3. 如果 $F(X)=X$，那么 X 是 F 的不动点。

对这些定义比较直观的理解就是把$\mathcal{U}$的元素当成某种语句或断言，以及把 F 的元素当成“认证”的关系，让它们在给出一组语句（前提）下能说明得出的是哪些新的语句（结论）。那么 F 闭集不能通过添加经 F 认证的元素而增大，因为它已经包含所有被其成员认证过的结论。F 一致集是“自我认证”的：其中每个断言都被当中的其他断言所认证。F 的不动点是一个既封闭又一致的集合：它包含了它的元素要求的所有认证，以及包含了所有元素得出的结论，除此之外别无其他。

21.1.3 例子：考虑下述含三个元素的集合$\mathcal{U}=\{a,b,c\}$的产生函数：

$$\begin{array}{lll} E_1(\emptyset) & = & \{c\} \\ E_1(\{a\}) & = & \{c\} \\ E_1(\{b\}) & = & \{c\} \\ E_1(\{c\}) & = & \{b,c\} \end{array} \qquad \begin{array}{lll} E_1(\{a,b\}) & = & \{c\} \\ E_1(\{a,c\}) & = & \{b,c\} \\ E_1(\{b,c\}) & = & \{a,b,c\} \\ E_1(\{a,b,c\}) & = & \{a,b,c\} \end{array}$$

其中只有一个 E_1 闭集$\{a,b,c\}$，4个 E_1 一致集$\emptyset,\{c\},\{b,c\},\{a,b,c\}$。

E_1 可以用一个推论的规则集来压缩表示：

$$\frac{}{c} \qquad \frac{c}{b} \qquad \frac{b \quad c}{a}$$

每个规则说明所有横线上面的元素为输入，而横线之下的则为输出。

21.1.4 定理［Knaster-Tarski（Tarski，1955）］：

1. 所有 F 封闭集的交集是最小的 F 不动点。
2. 所有 F 一致集的并集类型是最大的 F 不动点。

证明：这里只考虑第二点，第一点的证明是对称的。设 $C=\{X \mid X \subseteq F(X)\}$为所有 F 一致集的合集，设 P 为所有这些集合的并集。考虑到 F 是单调的，且对任何 $X\in C$ 有 X 是 F 一致的且 $X \subseteq P$，于是得出 $X \subseteq F(X) \subseteq F(P)$。因此 $P=\cup_{X\in C}X \subseteq F(P)$，即 P 是 F 一致的。而且，根据它的定义，P 最大的 F 一致集。再一次利用 F 的单调性，可得出 $F(P)\subseteq F(F(P))$。这就意味着，通过 C 的定义，可以得出 $F(P)\in C$。于是，对 C 的任意成员都有 $F(P)\subseteq P$，即 P 是 F 封闭的。现在已经证明 P 既是最大的 F 一致集又是 F 的不动点，所以 P 就是最大的不动点。

21.1.5 定义：F 的最小不动点记为 μF；F 的最大不动点记为 νF。

21.1.6 例子：从上面产生函数 E_1 的例子，可以得出 $\mu E_1=\nu E_1=\{a,b,c\}$。

21.1.7 练习［★］：假设全集$\{a,b,c\}$上的一个产生函数 E_2 由下面的推论规则定义：

$$\frac{}{a} \qquad \frac{c}{b} \qquad \frac{a \quad b}{c}$$

如对 E_1 所做的那样，请明确写出关系 E_2 的序对集。列出所有 E_2 封闭和 E_2 一致的集合。并指明 μE_1 和 νE_2 是哪些。

注意到 μF 本身是 F 封闭的(因此,它是最小的 F 闭集),νF 是 F 一致的(因此,它是最大的 F 一致集)。这个结论提供了一对最基本的推理工具:

21.1.8　推论[关于定理(21.1.4)]:

1. 归纳原则:如果 X 是 F 封闭的,那么 $\mu F \subseteq X$。
2. 共归纳原则:如果 X 是 F 一致的,那么 $X \subseteq \nu F$。

提出这两条原则的原因是想把集合 X 作为一个谓词,表示它的特征集合——谓词为真的 $\mathcal{U}$ 的子集;如果说元素 x 具有性质 X,则表示 x 在集合 X 中。现在,归纳原则说明特征集为 F 闭集的任何性质(也就是说,性质保持为 F 的)对归纳定义集合 μF 的所有元素都为真。

另外,共归纳原则提供了建立元素 x 在共归纳定义的集合 νF 中的方法。为了表明 $x \in \nu F$,需要找到集合 X,使 $x \in X$ 且 X 是 F 一致的。虽然共归纳原则比起归纳原则要让人感到陌生些,但是它却是计算机科学许多领域的中心。举例来说,它是基于互模拟的,并发理论的主要证明工具,而且它还处于许多模型检查算法的核心。

归纳和共归纳原则在全章中将被大量使用。我们不会根据产生函数和谓词来写出所有的归纳参数,而是为了简便,更多地使用熟悉的缩写形式,如结构化归纳。共归纳参数的表示会更为明确。

21.1.9　练习[推荐,★★★]: 证明自然数的一般归纳原则[参见公理(2.4.1)]及自然数对的字典序归纳原则[参见公理(2.4.4)]符合推论(21.1.8)的归纳原则。

21.2　有限类型和无穷类型

接下来我们将举例说明最大不动点的一般化定义和用具体子类型说明共归纳证明方法。在此之前,先明确地说明一下如何将类型视为(有限或无穷)树。

简便起见,本章仅考虑三种类型构造子:→, × 和 Top。我们将类型表示为树的形式(可能是无穷的),节点用→, × 和 Top 三种符号中的一种来作为标签。这种定义符合现在的需要;若对无穷标签树的一般处理方式感兴趣可参阅 Courcelle(1983)。

我们把 1 和 2 的序列集记为 $\{1,2\}^*$。空序列记为 •,而 i^k 代表 i 的 k 次方。如果 π 和 σ 都是序列,那么 π,σ 表示的是 π 和 σ 串联。

21.2.1　定义: 一个树类型[①] (或简称为树)是一个局部函数 $\mathrm{T} \in \{1,2\}^* \rightharpoonup \{\to, \times, \mathrm{Top}\}$,它满足下列约束条件:

- T(•) 有定义
- 如果 $\mathrm{T}(\pi,\sigma)$ 定义,那么 $\mathrm{T}(\pi)$ 也有定义。
- 如果 $\mathrm{T}(\pi) = \to$ 或者 $\mathrm{T}(\pi) = \times$,那么 $\mathrm{T}(\pi,1)$ 和 $\mathrm{T}(\pi,2)$ 有定义。
- 如果 $\mathrm{T}(\pi) = \mathrm{Top}$,那么 $\mathrm{T}(\pi,1)$ 和 $\mathrm{T}(\pi,2)$ 没有定义。

如果 $dom(\mathrm{T})$ 是有限的,那么树类型 T 也是有限的。所有树类型的集合可记为 $\mathcal{T}$;所有有限树类型的子集都记为 $\mathcal{T}_f$。

① "树类型"的提法有些不可靠,但它能使我们在 21.8 节中谈到递归类型作为一个含 μ("μ 类型")的有穷表达式时更直观些。

为了方便,将满足 T(•) = Top 的树 T 写为 Top。若 T_1 和 T_2 都是树,把满足($T_1 \times T_2$)(•) = × 和($T_1 \times T_2$)(i,π) = T_i(π)的树记为 $T_1 \to T_2$,把($T_1 \to T_2$)(•) = → 和($T_1 \to T_2$)(i,π) = T_i(π)的树记为 $T_1 \to T_2$,其中 i = 1,2。比如:(Top × Top)→Top 表示的就是由函数 T(•) = →,T(1) = × 和 T(2) = T(1,1) = T(1,2) = Top 定义的有限树类型 T。我们用省略号来表示非有限树类型。比如 Top →(Top →(Top →…))对应于类型 T,其中 T 的定义是对所有 $k \geqslant 0$,T(2^k) = →,且对所有 $k \geqslant 0$,T(2^k,1) = Top。图 21.1 描述了这些规定。

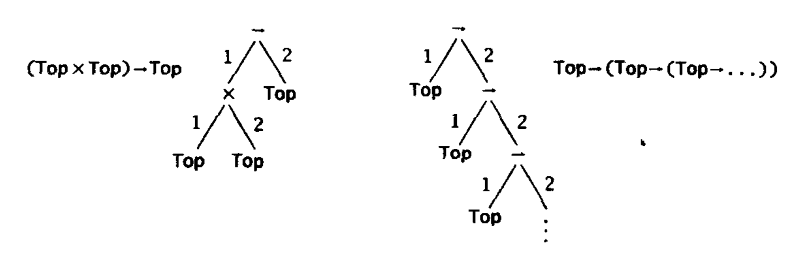


图 21.1 树类型例子

可以根据语法给出有限树类型更加简洁的定义形式:

$$
\begin{aligned}
\mathsf{T} \ ::= \ & \mathsf{Top} \\
& \mathsf{T} \times \mathsf{T} \\
& \mathsf{T} \to \mathsf{T}
\end{aligned}
$$

形式上,$\mathcal{T}_f$是用语法描述的产生函数的最小不动点。产生函数的全集是所有用→,× 和 Top 作为标签有限和无穷树的集合[也就是说,不考虑最后两个条件,对定义(21.2.1)一般化后的集合形式]。若用最大不动点代替最小不动点,相同的产生函数可推出整个集合$\mathcal{T}$。

21.2.2 练习[推荐,★★★]:根据前文提出的思想,假设有全集$\mathcal{U}$和产生函数 $F \in \mathcal{P}(\mathcal{U}) \to \mathcal{P}(\mathcal{U})$使得有限树类型集合$\mathcal{T}_f$是 F 的最小不动点,而所有树类型$\mathcal{T}$的集合是它的最大不动点。

21.3 子类型

我们对某全集上的单调函数的最小或最大不动点的树类型及有限树类型定义子类型关系。对有限树类型的子类型,全集指的是有限树序对集合$\mathcal{T}_f \times \mathcal{T}_f$;产生函数将该全集的子集(即$\mathcal{T}_f$上的关系)映射到其他的子集,且它们的不动点也将是$\mathcal{T}_f$上的关系。任意树(有限或无穷)的子类型,全集为$\mathcal{T} \times \mathcal{T}$。

21.3.1 定义[有限子类型]:对两个有限树类型 S 和 T,如果(S,T)$\in \mu S_f$,其中单调函数 $S_f \in \mathcal{P}(\mathcal{T}_f \times \mathcal{T}_f) \to \mathcal{P}(\mathcal{T}_f \times \mathcal{T}_f)$定义为:

$$
\begin{aligned}
S_f(R) \ = \ & \{(\mathsf{T}, \mathsf{Top}) \mid \mathsf{T} \in \mathcal{T}_f\} \\
\cup \ & \{(\mathsf{S}_1 \times \mathsf{S}_2, \mathsf{T}_1 \times \mathsf{T}_2) \mid (\mathsf{S}_1, \mathsf{T}_1), (\mathsf{S}_2, \mathsf{T}_2) \in R\} \\
\cup \ & \{(\mathsf{S}_1 \to \mathsf{S}_2, \mathsf{T}_1 \to \mathsf{T}_2) \mid (\mathsf{T}_1, \mathsf{S}_1), (\mathsf{S}_2, \mathsf{T}_2) \in R\}.
\end{aligned}
$$

那么 S 和 T 为子类关系(即“S 是 T 的一个子类型”)。根据如下的推论规则,可知该产生函数准确地反映了子类型关系的标准定义:

$$\frac{}{\mathsf{T} <: \mathsf{Top}}$$

$$\frac{\mathsf{S_1} <: \mathsf{T_1} \qquad \mathsf{S_2} <: \mathsf{T_2}}{\mathsf{S_1} \times \mathsf{S_2} <: \mathsf{T_1} \times \mathsf{T_2}}$$

$$\frac{\mathsf{T_1} <: \mathsf{S_1} \qquad \mathsf{S_2} <: \mathsf{T_2}}{\mathsf{S_1} \to \mathsf{S_2} <: \mathsf{T_1} \to \mathsf{T_2}}$$

其中第二个和第三个规则横线上边的语句 S <: T 可读为“如果序对(S,T)是 S_f 的输入参数”,而横线之下的部分读为“那么(S,T)就为结果”。

21.3.2　定义[无穷子类型化]:对两个树类型(有限或无穷)S 和 T,如果(S,T)∈ νS,其中 $S \in \mathcal{P}(\mathcal{T} \times \mathcal{T}) \to \mathcal{P}(\mathcal{T} \times \mathcal{T})$被定义为:

$$\begin{array}{rcl} S(R) & = & \{(\mathsf{T}, \mathsf{Top}) \mid \mathsf{T} \in \mathcal{T}\} \\ & \cup & \{(\mathsf{S_1} \times \mathsf{S_2}, \mathsf{T_1} \times \mathsf{T_2}) \mid (\mathsf{S_1}, \mathsf{T_1}), (\mathsf{S_2}, \mathsf{T_2}) \in R\} \\ & \cup & \{(\mathsf{S_1} \to \mathsf{S_2}, \mathsf{T_1} \to \mathsf{T_2}) \mid (\mathsf{T_1}, \mathsf{S_1}), (\mathsf{S_2}, \mathsf{T_2}) \in R\}. \end{array}$$

那么 S 和 T 为子类型关系。

注意该关系的推论规则表示方式与上面的归纳关系相同:所做的变动就是考虑一个更大类型全集,所用的是最大不动点而不是最小不动点。

21.3.3　练习[★]:通过列举不在 νS 中的序对(S,T),验证一下 νS 并不是全部的 $\mathcal{T} \times \mathcal{T}$。

21.3.4　练习[★]:是否存在与 νS 有关而与 μS 无关一个类型序对(S,T)? 与 νS_f 有关而与 μS_f 无关的类型序对(S,T)存在吗?

无穷树类型上的子类型关系一个基本属性,即传递性,可立即被证明(在 16.1 节中已知有限类型的子类型具有传递性)。如果子类型关系不具传递性,那么求值中的类型保持关键特性将马上失败。假设有类型 S,T 和 U,其中 S <: T,T <: U,但不满足 S <: U。设 s 是类型 S 的一个值,f 为类型 U→Top 的一个函数,那么对每个应用使用一次包含规则,项(λx :T.fx)s 有可能是可类型化的,但是实际上它一步就会得出不良类型 f s。

21.3.5　定义:如果 R 在单调函数 $TR(R) = \{(x,y) \mid \exists z \in \mathcal{U}$ 且 $(x,z),(z,y) \in R\}$ 下是封闭的(即 $TR(R) \subseteq R$),那么关系 $R \subseteq \mathcal{U} \times \mathcal{U}$ 有传递性。

21.3.6　引理:设 $F \in \mathcal{P}(\mathcal{U} \times \mathcal{U}) \to \mathcal{P}(\mathcal{U} \times \mathcal{U})$为一单调函数。如果对任意 $R \subseteq \mathcal{U} \times \mathcal{U}$ 都有 $TR(F(R)) \subseteq (F(TR(R))$,那么 νF 是可传递的。

证明:因为 νF 是一个不动点,$\nu F = F(\nu F)$就表示 $TR(\nu F) = TR(F(\nu F))$。这样根据引理的假设有 $TR(\nu F) \subseteq F(TR(\nu F))$。换言之,$TR(\nu F)$是 F 一致的,因此,根据共归纳原则,$TR(\nu F) \subseteq \nu F$。等价地,根据定义(21.3.5)$\nu F$ 有传递性。

这条引理会令人回想起推论系统中传递规则的传统冗余处理技术,通常称为“cut 消去证明”(参见 16.1 节)。条件 $TR(F(R)) \subseteq F(TR(R))$对应于技术中至关重要的一步:应用 F 中的规则,然后运用 TR 传递性的规则,从 R 的某些语句中可得出一个特定的语句,而换个方式,我们认为这条语句还可以通过相反的步骤得到——首先运用 TR 传递性的规则,然后才是 F 的规则。运用这个引理来建立子类型关系的传递性。

21.3.7 定理：νS 具有传递性。

证明：根据引理(21.3.6)，可以证明对任意 $R \subseteq \mathcal{T} \times \mathcal{T}$，都满足 $TR(S(R)) \subseteq S(TR(R))$。设(S,T)∈ $TR(S(R))$。根据 TR 的定义，存在某个 U∈$\mathcal{T}$ 使得(S,U)，(U,T) ∈$S(R)$。我们的目的就是证明(S,T)∈$S(TR(R))$。下面考虑一下 U 的可能形式：

情况： U = Top

因为(U,T)∈$S(R)$，S 的定义表明 T 必为 Top。但是对于任意 A 和 Q 都有(A,Top)∈$S(Q)$；特殊的情况为，(S,T) = (S,Top)∈$S(TR(R))$。

情况： $U = U_1 \times U_2$

如果 T = Top，那么就如前一种情况一样(S,T)∈$S(TR(R))$。否则(U,T)∈$S(R)$意味着 $T = T_1 \times T_2$，其中，(U_1, T_1)，$(U_2, T_2) \in R$。同理，(S,U)∈$S(R)$说明 $S = S_1 \times S_2$，其中，(S_1, U_1)，$(S_2, U_2) \in R$。根据 TR 的定义，我们可以从 S 的定义推导出的$(S_1 \times S_2, T_1 \times T_2) \in S(TR(R))$中得出$(S_1, T_1)$，$(S_2, T_2) \in TR(R)$。

情况： $U = U_1 \to U_2$

同理可证。

21.3.8 练习[推荐，★★★]：证明无穷树类型的子类型关系也具有自反性。

在下一节中，将通过对有限类型的标准子类型化的处理方法和现有的无穷树类型的子类型化方法的比较，继续讨论传递性。第一次阅读时，可以跳过本节或只做浏览。

21.4 传递性的偏离

第 16 章讲到了归纳定义子类型关系的标准形式有两种：为提高可读性而采用的声明性表示方式和或多或少面向实现的算法表示方式。在简单的系统中，这两种表示方式基本上相似；但对更复杂的系统，它们就完全不一样了，若要证明它们定义的是类型上相同的关系，将是一件极具挑战性的事(第 28 章将就这一点举了个例子；其他的都已经学到过)。

声明性表示与算法表示之间一个最突出的不同点就是：声明性表示中包含了一个明确的传递性规则(如果 S <: U 且 U <: T，那么 S <: T)，而算法系统却不是这样。该规则在算法系统中根本用不上，因为以面向目标的方式总会引起对 U 的猜测。

在声明性系统中，传递性规则起了两个重要的作用：(1)它使读者明白子类型关系的确是传递性的；(2)传递性使其他的规则处于一种更简单、更原始的形式，而在算法表示中，这些简单的规则必须穷尽所有的组合方式以形成复杂的数以百万条的规则，这样才能将每种可能出现的情况考虑清楚。例如，有了传递性规则，记录字段内的“深度子类型化”规则、增加新字段的“宽度子类型化”规则和字段的“置换”规则都可以单独描述，以便于理解(参见 15.2 节)。若不用传递性规则，这 3 条规则必须合并为一条一次可以完成宽度、深度和置换功能的规则，如在 16.1 节中所见。

还有一点令人感到惊讶的是，在声明性表示中加入了传递性规则可能会造成类似于归纳定义(不包括共归纳)带来的技巧上的影响。原因是，传递性其实是一个闭包的性质，它要求传递规则下的子类型关系是封闭的。因为有限类型的子类型关系本身定义为规则集的闭集形

式,所以可以很自然地将传递的封闭性用到其他的规则中。这形成了归纳定义和闭包性质的一般化性质:当归纳地将两个规则集合并时,会产生各自规则所封闭的最小关系。利用产生函数可将该结论进一步抽象化、形式化。

21.4.1　命题:设 F 和 G 为单调函数,且 $H(X)=F(X)\cup G(X)$,则 μH 为满足 F 封闭和 G 封闭的最小集合。

证明:(1)说明 μH 在 F 和 G 均为封闭的。根据定义知 $\mu H=H(\mu H)=F(\mu H)\cup G(\mu H)$,所以有 $F(\mu H)\subseteq\mu H$ 和 $G(\mu H)\subseteq\mu H$;(2)还要说明 μH 是 F 和 G 下的最小封闭集合。设存在集合 X 使得 $F(X)\subseteq X$ 且 $G(X)\subseteq X$,那么有 $H(X)=F(X)\cup G(X)\subseteq X$,也就是说 X 是 H 封闭的。因为 μH 是最小的 H 闭集(根据 Knaster-Tarski 定理),所以有 $\mu H\subseteq X$。

遗憾的是,这种传递封闭性的处理技巧对共归纳定义不起作用。如下的练习可说明,在产生共归纳定义关系的规则中添加传递性规则往往会得出一种退化的关系。

21.4.2　练习[★]:假设 F 是全集 $\mathcal{U}$ 中的一个产生函数。证明产生函数:

$$F^{TR}(R)=F(R)\cup TR(R)$$

的最大不动点 νF^{TR} 是 $\mathcal{U}\times\mathcal{U}$ 上的全关系。

所以谈到共归纳时,我们不考虑声明性表示,而只讨论算法表示。

21.5　成员检查

现在把话题转向本章的中心:给定某全集 $\mathcal{U}$ 上的产生函数 F 和元素 $x\in\mathcal{U}$,该如何判断 x 是否会落在 F 的最大不动点中。对最小不动点的成员检查将更简捷(参见练习 21.5.13)。

通常一个元素 $x\in\mathcal{U}$ 可通过很多方法由 F 产生。也就是说,可以有不止一个集合 $X\subseteq\mathcal{U}$,满足 $x\in F(X)$。称这种集合 X 为 x 的一个产生集。由于 F 的单调性,任何 x 的产生集的超集也是 x 的一个产生集,因此,将注意力放在最小的产生集上有一定的意义。更进一步,要把注意力放在一簇"可逆"的产生函数上,其中每个 x 最多只有一个最小产生集。

21.5.1　定义:如果对所有的 $x\in\mathcal{U}$,簇集:

$$G_x=\{X\subseteq\mathcal{U}\mid x\in F(X)\}$$

要么为空,要么包含一个惟一的为其他所有集合的子集的元素,则该产生函数 F 称为可逆的。当 F 可逆时,部分函数 $support_F\in\mathcal{U}\to\mathcal{P}(\mathcal{U})$ 定义如下:

$$support_F(x)=\begin{cases}X & \text{if } X\in G_x \text{ and } \forall X'\in G_x.\ X\subseteq X'\\ \uparrow & \text{if } G_x=\varnothing\end{cases}$$

support 函数可以写成集合的形式:

$$support_F(X)=\begin{cases}\bigcup_{x\in X} support_F(x) & \text{if } \forall x\in X.\ support_F(x)\downarrow\\ \uparrow & \text{otherwise}\end{cases}$$

如果 F 在上下文中是明确的,那么可以将 $support_F$ 下方的 F 省略(类似于后面定义到的基于 F 的函数)。

21.5.2 练习[★★]:证明定义(21.3.1)和定义(21.3.2)中的子类型关系的产生函数 S_f 和 S 是可逆的,并给出它们的支持函数。

我们的目标是提出一个检查产生函数 F 的最大和最小不动点中成员的算法。算法中最基本的一步包括"逆向执行 F":为了检查元素 x 的成员关系,需要得知 x 是如何从 F 中产生的。F 可逆的优势就是最多只有一种产生 x 的方式。而对于不可逆的 F,x 可通过多种方式产生,这样会造成算法所经过的路径形成组合爆炸。所以从现在起,仅限于讨论可逆的产生函数。

21.5.3 定义:假如 $support_F(x)\downarrow$,那么元素 x 就是 F 支持的,否则就是 F 不支持的。如果 $support_F(x)=\varnothing$,那么 F 支持的元素被称为 F 基。

注意,因为对每个 X,基 x 都在 $F(X)$中,所以对任意 X,不支持元素 x 不出现在 $F(X)$中。

一个可逆函数可以用支持图来形象化表示。例如,图 21.2 定义了全集$\{a,b,c,d,e,f,g,h,i\}$上的函数 E,用来指明哪些元素需要支持给定全集中的元素:对给定的 x,集合 $support_E(x)$包含所有 y,满足存在从 x 指向 y 的箭头。不支持元素用带短斜线的圆圈表示。在这个例子中,i 是惟一的不支持元素,而 g 是惟一基元素(注意:根据我们的定义,h 是支持的,即使它的支持集包括了一个不支持的元素)。

21.5.4 练习[★]:如例 21.1.3 中所示,给出该函数相应的推论规则。验证 $E(\{b,c\})=\{g,a,d\}$,$E(\{a,i\})=\{g,h\}$,及在图中表明为 μE 和 νE 的部分,确实为 E 的最小和最大的不动点。

观察图 21.2 可知,当且仅当支持图中不存在 x 可达的不支持元素,则元素 x 为最大不动点。这表明检查 x 是否在 νF 中的算法策略:列举出所有通过支持函数从 x 出发可达的元素;如果其中含有不支持元素,那么 x 就不在 νF 中;否则,x 在 νF 中。注意到,尽管图中的元素之间存在一些可达的循环,列举过程中要注意不要陷入死循环。在接下来的部分中,仍要注意这一点。

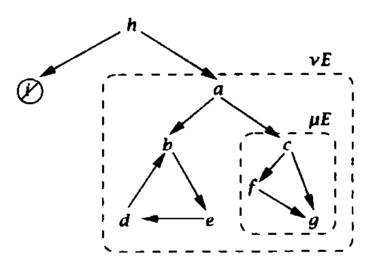


图 21.2 支持函数

21.5.5 定义:假设 F 是一个可逆的产生函数。定义布尔函数 gfp_F(或简写为 gfp)如下[①]:

$$
\begin{aligned}
gfp(X) \quad = \quad & \text{if } support(X)\uparrow, \text{ then } false \\
& \text{else if } support(X)\subseteq X, \text{ then } true \\
& \text{else } gfp(support(X)\cup X).
\end{aligned}
$$

① 这里对递归函数的定义采用标准的符号,即 gfp 是满足等式的最小偏函数。该函数本身可看成是合适的产生函数的最小不动点。任何指称语义的文章都对该函数进行了详细讨论,如 Gunter(1992),Winskel(1993)或 Mitchell(1996)。

直观地讲，*gfp* 从 X 出发，不断用支持函数强化它，直到它变得一致或者找到一个不支持元素。通过公式 $gfp(x) = gfp(\{x\})$ 可将 *gfp* 扩展为个别元素。

21.5.6 练习[★]：从图 21.2 中还可得出的结论，就是 νF 的元素 x 出现在支持图的循环中，那么它不是 μF 的元素（或者存在一条从 x 通往循环中其他元素的路径）。反之也成立吗——也就是说，如果元素 x 是 νF 的成员而不是 μF 的成员，那么它也可能有通往循环的途径吗？

接下来的部分将致力于证明 *gfp* 的正确性和可终止性（第一次阅读的人可以跳过这些内容而直接读下一节）。我们从 *support* 函数的一些特性开始讨论。

21.5.7 引理：当且仅当 $support_F(X)\downarrow$ 且 $support_F(X) \subseteq Y$，$X \subseteq F(Y)$ 成立。

证明：需要证明，当且仅当 $support(x)\downarrow$ 且 $support(x) \subseteq Y$ 时 $x \in F(Y)$。首先假设 $x \in F(Y)$。于是 $Y \in G_x = \{X \subseteq \mathcal{U} \mid x \in F(X)\}$，即 $G_x \neq \varnothing$。因为 F 可逆，所以 G_x 中的最小集 $support(x)$ 存在，且 $support(x) \subseteq Y$。反之，根据单调性，如果 $support(x) \subseteq Y$，则 $F(support(x)) \subseteq F(Y)$。但是根据支持函数的定义知 $x \in F(support(x))$，所以 $x \in F(Y)$。

21.5.8 引理：假设 P 是 F 的一个不动点。那么 $X \subseteq P$ 当且仅当 $support_F(X)\downarrow$ 且 $support_F(X) \subseteq P$。

证明：根据 $P = F(P)$，并运用引理(21.5.7)即可得证。

现在我们可以证明 *gfp* 的部分正确性（我们还没有考虑到它的完全正确性，因为一些产生函数会使 *gfp* 产生某些歧义。在本节的后一部分将证明产生函数的受限类的可终止性）。

21.5.9 定理：

1. 如果 $gfp_F(X) = \text{true}$，那么 $X \subseteq \nu F$。
2. 如果 $gfp_F(X) = \text{false}$，那么 $X \not\subseteq \nu F$。

证明：通过递归结构运行算法的归纳来证明每个命题。

1. 从 *gfp* 的定义中很容易得知有两种 $gfp(X)$ 返回真值的情况。如果 $gfp(X) = true$ 是由于 $support(X) \subseteq X$，那么根据引理(21.5.7)，有 $X \subseteq F(X)$，即 X 是 F 一致的；这样根据共归纳原则可知 $X \subseteq \nu F$。另一方面，如果 $gfp(X) = true$ 是由于 $gfp(support(X) \cup X) = true$，那么由归纳假设，有 $support(X) \cup X \subseteq \nu F$，所以 $X \subseteq \nu F$。
2. 同样，也有两种使 $gfp(X) = false$ 的方法。假设第一种 $gfp(X) = false$ 是因为 $support(X)\uparrow$。那么根据引理(21.5.8)可知 $X \not\subseteq \nu F$。另一种假设使 $gfp(X) = false$ 的原因是 $gfp(support(X) \cup X) = false$。根据归纳假设，有 $support(X) \cup X \not\subseteq \nu F$。等价地，$X \not\subseteq \nu F$ 或 $support(X) \not\subseteq \nu F$。无论哪种都可得出 $X \not\subseteq \nu F$[第二种情况还要根据引理(21.5.8)]。

接下来我们确定 *gfp* 的一个完全终止的条件，给出一类产生函数使它的算法能够保证终止。为了描述这个类，我们还需要一些术语。

21.5.10 定义:给定可逆产生函数 F 和元素 $x\in\mathcal{U}$,x 的直接前驱集合 $pred_F(x)$(或简写为 $pred(x)$)为:

$$pred(x) = \begin{cases} \varnothing & \text{if } support(x)\uparrow \\ support(x) & \text{if } support(x)\downarrow \end{cases}$$

对集合 $X\subseteq\mathcal{U}$的扩展定义为:

$$pred(X) = \bigcup_{x\in X} pred(x).$$

通过支持函数从集合 X 出发可达的所有元素的集合 $reachable_F(X)$(或简写为 $reachable(X)$)为:

$$reachable(X) = \bigcup_{n\geq 0} pred^n(X).$$

对单个元素 $x\in\mathcal{U}$的扩展定义为:

$$reachable(x) = reachable(\{x\}).$$

如果 $y\in reachable(x)$,那么元素 $y\in\mathcal{U}$从 x 出发是可达的。

21.5.11 定义:如果对任一 $x\in\mathcal{U}$,$reachable(x)$是有限的,那么可逆产生函数 F 也可称为是有限状态的。

对有限状态的产生函数而言,因为 *gfp* 的搜索空间有限,所以 *gfp* 通常都可以终止。

21.5.12 定理:如果 $reachable_F(X)$是有限的,那么 $gfp_F(X)$就可以定义。因此,如果 F 是有限状态,那么对任意有限集 $X\subseteq U$,$gfp_F(X)$都可终止。

证明:对原有的 $gfp(X)$产生的调用图中的每个递归调用 $gfp(Y)$,可得到 $Y\subseteq reachable(X)$。而且,每次调用 Y 都严格递增。由于 $reachable(X)$有限,$m(Y) = |reachable(X)| - |Y|$ 可作为衡量 *gfp* 可终止的手段。

21.5.13 练习[★★★]:假设 F 是一可逆产生函数,函数 lfp_F(或简写为 *lfp*)定义如下:

$$\begin{aligned} lfp(X) \quad = \quad & \text{if } support(X)\uparrow\text{, then } false \\ & \text{else if } X = \varnothing\text{, then } true \\ & \text{else } lfp(support(X)). \end{aligned}$$

直观上说,*lfp* 从集合 X 开始,利用支持关系约简它,直至为空。根据:

1. 如果 $lfp_F(X) = true$,那么 $X\subseteq\mu F$。

2. 如果 $lfp_F(X) = false$,那么 $X\not\subseteq\mu F$。

这个算法被证明是部分正确的。你能否找到一簇产生函数,以保证 lfp_F 对所有有限输入都能终止?

21.6 更高效算法

尽管 *gfp* 算法是正确的,但是它的效率却并不高,因为在每次进行递归调用时,它都要重新计算整个集合 X 的支持函数。比如在图 21.2 中,函数 E 的 *gfp* 的踪迹:

$$
\begin{aligned}
& gfp(\{a\}) \\
= \; & gfp(\{a,b,c\}) \\
= \; & gfp(\{a,b,c,e,f,g\}) \\
= \; & gfp(\{a,b,c,e,f,g,d\}) \\
= \; & true.
\end{aligned}
$$

显然，$support(a)$被计算了 4 次。可以保持假设集 A（它的支持集已经被考虑了）和目标集 X（它的支持集未被考虑），这样就能减少计算次数从而改进算法。

21.6.1　定义：假设 F 为一可逆的产生函数。定义函数 gfp^a_F（或简写为 gfp^a）如下（其中上标 a 代表 assumptions）：

$$
\begin{aligned}
gfp^a(A,X) \;=\; & \text{if } support(X)\uparrow, \text{ then } false \\
& \text{else if } X = \varnothing, \text{ then } true \\
& \text{else } gfp^a(A\cup X,\; support(X)\setminus(A\cup X)).
\end{aligned}
$$

为了检查 $x\in \nu F$，应计算 $gfp^a(\varnothing,\{x\})$。

此算法（如本节中的后两个算法）计算每个元素的支持函数至多一次。前一例子的计算踪迹为：

$$
\begin{aligned}
& gfp^a(\varnothing, \{a\}) \\
= \; & gfp^a(\{a\}, \{b,c\}) \\
= \; & gfp^a(\{a,b,c\}, \{e,f,g\}) \\
= \; & gfp^a(\{a,b,c,e,f,g\}, \{d\}) \\
= \; & gfp^a(\{a,b,c,e,f,g,d\}, \varnothing) \\
= \; & true.
\end{aligned}
$$

很显然，此算法比前一节我们看到的算法稍微精巧一些。

21.6.2　定理：

1. 如果 $support_F(A)\subseteq A\cup X$ 且 $gfp^a_F(A,X)=true$，则有 $A\cup X\subseteq \nu F$。
2. 如果 $gfp^a_F(A,X)=false$，则有 $X\nsubseteq \nu F$。

证明：同定理(21.5.9)。

本节余下的部分将检查 gfp 算法的两种改进算法，使它们与递归类型熟知的子类型算法更为接近。第一次阅读的人可以直接跳到下一节。

21.6.3　定义：对 gfp^a 的小小变化是从 X 中一次选一个元素并扩展它的支持函数后的算法。新的算法被称为 gfp^s_F（或简写为 gfp^s，s 代表 single）：

$$
\begin{aligned}
gfp^s(A,X) \;=\; & \text{if } X=\varnothing, \text{ then } true \\
& \text{else let } x \text{ be some element of } X \text{ in} \\
& \quad \text{if } x\in A \text{ then } gfp^s(A, X\setminus\{x\}) \\
& \quad \text{else if } support(x)\uparrow \text{ then } false \\
& \quad \text{else } gfp^s(A\cup\{x\},\; (X\cup support(x))\setminus(A\cup\{x\})).
\end{aligned}
$$

此算法的正确性（比如，递归“环”的不变性）与定理(21.6.2)完全相同。

与上面算法不同的是，许多递归子类型化算法用一个备选元素，而不是一个集合作为参

数。对上面的算法做小小的改动能使之更接近于这些算法。修改的算法不再是尾递归[①],因为它用调用栈来保存还未检查的下一个目标。另一个变化是,算法将假设集 A 作为一个参数,并返回一个新的假设集作为结果。这就允许它在全递归调用中记录下已经产生的子类型化假设,并在后面的调用中重用。效果上,假设集从算法名 *gfpt* 开始通过递归调用图进行整理。

21.6.4 定义:给定一可逆产生函数 F,定义 gfp^t_F(或简写为 gfp^t)如下:

$$
\begin{array}{lcl}
gfp^t(A,x) & = & \text{if } x \in A, \text{ then } A \\
& & \text{else if } support(x) \uparrow, \text{ then } fail \\
& & \text{else} \\
& & \quad \text{let } \{x_1,\cdots,x_n\} = support(x) \text{ in} \\
& & \quad \text{let } A_0 = A \cup \{x\} \text{ in} \\
& & \quad \text{let } A_1 = gfp^t(A_0,x_1) \text{ in} \\
& & \quad \cdots \\
& & \quad \text{let } A_n = gfp^t(A_{n-1},x_n) \text{ in} \\
& & \quad A_n.
\end{array}
$$

为了检查 $x\in \nu F$,应计算 $gfp^t(\varnothing, x)$。如果调用成功,则 $x\in \nu F$,否则,$x \notin \nu F$。如果失败我们有如下约定:如果表达式 B 失败,那么"let $A = B$ in C"也失败。这就避免了对每个 gfp^t 的递归调用明确写明"异常处理"的子句。

此算法正确的语句必须在以前的基础上进行修改,通过给元素(其支持函数仍有待检查)设定一个额外的"栈"X,以此来考虑该公式的非尾递归特性。

21.6.5 引理:

1. 如果 $gfp^t_F(A,x) = A'$,那么 $A\cup\{x\}\subseteq A'$。
2. 对所有 X,如果有 $support_F(A)\subseteq A\cup X\cup\{x\}$ 且 $gfp^t_F(A,x) = A'$,那么 $support_F(A')\subseteq A'\cup X$。

证明:第(1)部分是对运行算法递归结构的常规归纳。
第(2)部分也是通过对运行算法递归结构的归纳。如果 $x\in A$,那么 $A' = A$,且由假设可立即得出想要的结论。另一方面,如果 $A'\neq A$,而且还考虑特殊情况:$support(x)$ 含有两个元素 x_1 和 x_2——一般的情况(此处不证明)证明原理相似,对 $support(x)$ 的长度进行内部归纳。算法计算 A_0, A_1 和 A_2,并返回 A_2。对任意 X_0,我们想要证明:如果 $support(A)\subseteq A\cup\{x\}\cup X_0$,那么 $support(A_2)\subseteq A_2\cup X_0$。设 $X_1 = X_0\cup\{x_2\}$。因为:

$$
\begin{array}{lcl}
support(A_0) & = & support(A)\cup support(x) \\
& = & support(A)\cup\{x_1,x_2\} \\
& \subseteq & A\cup\{x\}\cup X_0\cup\{x_1,x_2\} \\
& = & A_0\cup X_0\cup\{x_1,x_2\} \\
& = & A_0\cup X_1\cup\{x_1\},
\end{array}
$$

① 尾递归调用(或称为尾调用)指一个递归调用,是调用函数的最后行为,即递归调用返回的结果也是调用者的结果。尾调用很有意思,因为大部分函数语言的编译器都把尾调用当成一个小的分支,通过重用调用者的栈空间而不是为递归调用分配一个新栈。这意味着尾递归调用循环将被编译为与 while 循环等价的机器代码。

通过将把 X 实例化为 X_1，可以对第一次递归调用归纳假设。这就得出 $support(A_1) \subseteq A_1 \cup X_1 = A_1 \cup \{x_2\} \cup X_0$。现在我们可以通过把 X 实例化为 X_0，对第二次递归调用进行归纳假设，并得出所要的结果：$support(A_2) \subseteq A_2 \cup X_0$。

21.6.6 定理：

1. 如果 $gfp^t_F(\varnothing, x) = A'$，那么 $x \in \nu F$。
2. 如果 $gfp^t_F(\varnothing, x) = fail$，那么 $x \notin \nu F$。

证明：第(1)部分，用引理[21.6.5(1)]，得出 $x \in A'$。对引理[21.6.5(2)]用 $X = \varnothing$ 实例化，得出 $support(A') \subseteq A'$，即根据引理(21.5.7)，A' 是 F 一致的，因此由共归纳还有 $A' \subseteq \nu F$。对第(2)部分，我们认为[通过引理(21.5.8)]，对 gfp^t_F 算法的深度进行简单归纳了对某些 A，如果 $gfp^t_F(A, x) = fail$，那么 $x \notin \nu F$。

因为本节中所有的算法都要检查可达集（reachable set），所有算法的终止条件都与原始算法的条件相同：当 F 有限时，对所有的输入都能终止。

21.7 正则树

此时，我们已经将检查集合中成员关系的通用算法定义为产生函数 F 最大不动点的集合，并假设 F 是可逆且有限的；另外，还说明了如何将无穷树之间的子类型化定义为一个特殊产生函数 S 的最大不动点。下一步显然就是用 S 将其中一个算法实例化。当然，该具体的算法不会对所有的输入都终止，因为通常给定的一对无穷类型的可达状态集合可以是无穷的。但是，将在本节中看到，如果只考虑某种良行为形式上的无穷类型，称为正则树，那么可达状态的集合将确定为有限的，并且子类型的检查算法也将是可以终止的。

21.7.1 定义：
对树类型 S 和 T，如果对某 π 有 $S = \lambda\sigma.T(\pi,\sigma)$ ——也就是说，对从路径到符号的函数 S，如果能够通过给 T 的参数路径添加某个常量前缀 π 来得到，则 S 是 T 的一棵子树；前缀 π 相当于从 T 的根到 S 的根路径。我们把所有子树 T 的集合写为 $subtrees$(T)。

21.7.2 定义：如果 $subtrees$(T)是有限的，即 T 有有限多的不同子树，那么树类型 $T \in \mathcal{T}$ 就是正则的。正则树类型的集合写为 $\mathcal{T}_r$。

21.7.3 例子：

1. 任意有限树类型都是正则的；不同子树的数目最多为节点的数目。树类型的不同子树的数目严格小于节点数目。比如，T = Top→(Top × Top)具有 5 个节点但只有 3 棵不同的子树（T，Top × Top 和 Top）。
2. 某些无穷树类型是正则的。比如树：

   T = Top × (Top × (Top × ···))

仅有 2 棵不同的子树（T 和 Top）。

3. 树类型：

   T = B× (A× (B× (A× (A× (B× (A× (A× (A× (B× ···)

当连续的每对 B 被逐渐增多的 A 分隔后，它就不是正则的。因为 T 为非正则的，所以包含子类型序对的集合 $reachable_S$(T,T)需要证明 T <: T 是无穷的。

21.7.4　命题：产生函数 S 对正则树的限定函数 S_r 是有限的。

证明：需要证明对正则树类型的任意序对(S,T)，集合 $reachable_{S_r}$(S,T)都是有限的。注意到 $reachable_{S_r}(S,T) \subseteq subtrees(S) \times subtrees(T)$；因为 $subtrees$(S)和 $subtrees$(T)都是有限的，$subtrees(S) \times subtrees(T)$也是有限的。

这就意味着可以通过将某个成员算法用 S 实例化后得到正则树类型上的子类关系的一个判定过程。自然地，在实际的执行中，正则树必须要由一些有限结构来表示。其中一种表示方法——μ 符号，将在下一节中讨论。

21.8　μ 类型

本节讨论有限 μ 符号，定义 μ 表达式上的子类型化，并将这种子类型与树类型的子类型联系起来。

21.8.1　定义：设 X 为一个类型变量的可数集$\{X_1, X_2, \cdots\}$。原始 μ 类型的集合 $\mathcal{T}^{raw}_m$ 依据如下语法定义：

$$
\begin{aligned}
T \; ::= \; & X \\
& Top \\
& T \times T \\
& T \to T \\
& \mu X.T
\end{aligned}
$$

语法算子 μ 是一个绑定器，它以标准的方式将囿变量和自由变量，封闭原始 μ 类型及与原始 μ 类型等价的概念以囿变量的形式重命名。用 FV(T)来表示原始 μ 类型的自由变量集。在原始 μ 类型 T 中，对 X 自由出现的地方，照样可以定义避免捕获代换$[X \mapsto S]T$。

原始 μ 类型必须稍加限制以获得与正则树紧密的对应：我们希望能够用 μ 类型的无穷展开的形式很快读出树类型，但是仍存在某些无法可靠地解释为树类型表示方式的原始 μ 类型。这些类型有形如 $\mu X.\mu X_1 \cdots \mu X_n.X$，的子表达式，其中从 X_1 到 X_n 都与 X 不同。举例来讲，对 $T = \mu X.X$，T 后展开再次出现新的 T，所以不能通过 T 的展开来通读树。因此提出了如下定义。

21.8.2　定义：如果对于任一形如 $\mu X.\mu X_1 \cdots \mu X_n.S$ 的 T 的子表达式(其中 S 不是 X)，原始 μ 类型 T 是收缩的。等价地，如果类型体中每个 μ 囿变量的出现都是用→或 × 与其绑定器分隔开的，则该原始 μ 类型为收缩的。

如果原始 μ 类型是收缩的，那么可直接简称为 μ 类型。μ 类型的集合记为 $\mathcal{T}_m$。

当 T 是一个 μ 类型时，在 T 前用 μ-$height$(T)来表示 μ 绑定的次数。

通常将 μ 类型作为无穷正则树的有限概念表示形式，定义如下：

21.8.3　定义：函数 $treeof$，将闭 μ 类型映射为树类型，归纳定义如下：

$$
\begin{array}{lcl}
treeof(\mathsf{Top})(\bullet) & = & \mathsf{Top} \\
treeof(\mathsf{T_1}\to\mathsf{T_2})(\bullet) & = & \to \\
treeof(\mathsf{T_1}\to\mathsf{T_2})(i,\pi) & = & treeof(\mathsf{T}_i)(\pi) \\
treeof(\mathsf{T_1}\times\mathsf{T_2})(\bullet) & = & \times \\
treeof(\mathsf{T_1}\times\mathsf{T_2})(i,\pi) & = & treeof(\mathsf{T}_i)(\pi) \\
treeof(\mu\mathsf{X.T})(\pi) & = & treeof([\mathsf{X}\mapsto\mu\mathsf{X.T}]\mathsf{T})(\pi)
\end{array}
$$

为了证明定义可靠性(即可终止的),应注意如下:

1. 对右端 *treeof* 进行递归可减少序对($|\pi|$,μ-*height*(T))的词典长度:S→T 和 S×T 的情况会减少$|\pi|$,而 μX.T 保留了$|\pi|$但减少了 μ-*height*(T)。
2. 所有的递归调用都会保持参数类型的收缩性和闭包性。尤其是当且仅当类型 μX.T 的展开式[X ↦ μX.T]T 为收缩且封闭时,μX.T 也为收缩且封闭的。这就为 *treeof*(μX.T)定义的展开做出了判断。

可在 *treeof* 函数中加入类型序对(S,T),并定义 *treeof*(S,T) = (*treeof*(S),*treeof*(T))。

将 *treeof* 应用到 μ 类型的例子如图 21.3 所示。

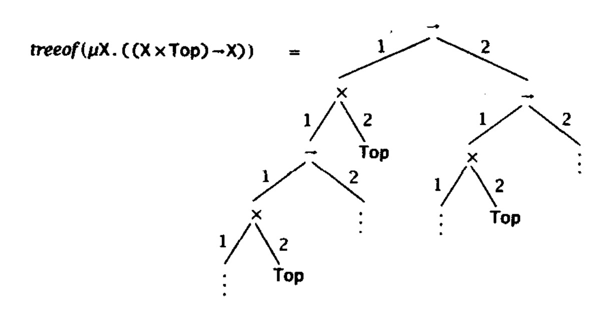


图 21.3　*treeof* 应用举例

树类型的子类型关系已经在 21.3 节中作为产生函数 S 的最大不动点定义出来。在本节中,我们用 μ 类型扩展类型语法,它的行为能够用 μ-folding 规则直接描述出来:

$$\frac{\mathsf{S} <: [\mathsf{X}\mapsto\mu\mathsf{X.T}]\mathsf{T}}{\mathsf{S} <: \mu\mathsf{X.T}} \quad \text{和} \quad \frac{[\mathsf{X}\mapsto\mu\mathsf{X.S}]\mathsf{S} <: \mathsf{T}}{\mu\mathsf{X.S} <: \mathsf{T}}$$

形式上,通过给出一产生函数 S_m 来定义 μ 类型的子类型,用三个和 S 定义相同的子句,还有两个附加的对应 μ-folding 规则的子句。

21.8.4　定义:对两个 μ 类型 S 和 T,如果满足(S,T)∈ νS_m,其中单调函数 $S_m \in \mathcal{P}(\mathcal{T}_m \times \mathcal{T}_m) \to \mathcal{P}(\mathcal{T}_m \times \mathcal{T}_m)$,的定义为:

$$
\begin{array}{rcl}
S_m(R) & = & \{(\mathsf{S},\mathsf{Top}) \mid \mathsf{S}\in\mathcal{T}_m\} \\
& \cup & \{(\mathsf{S_1}\times\mathsf{S_2},\mathsf{T_1}\times\mathsf{T_2}) \mid (\mathsf{S_1},\mathsf{T_1}),\ (\mathsf{S_2},\mathsf{T_2})\in R\} \\
& \cup & \{(\mathsf{S_1}\to\mathsf{S_2},\mathsf{T_1}\to\mathsf{T_2}) \mid (\mathsf{T_1},\mathsf{S_1}),\ (\mathsf{S_2},\mathsf{T_2})\in R\} \\
& \cup & \{(\mathsf{S},\mu\mathsf{X.T}) \mid (\mathsf{S},[\mathsf{X}\mapsto\mu\mathsf{X.T}]\mathsf{T})\in R\} \\
& \cup & \{(\mu\mathsf{X.S},\mathsf{T}) \mid ([\mathsf{X}\mapsto\mu\mathsf{X.S}]\mathsf{S},\mathsf{T})\in R,\ \mathsf{T}\neq\mathsf{Top},\ \text{and}\ \mathsf{T}\neq\mu\mathsf{Y.T_1}\}.
\end{array}
$$

则 S 和 T 为子类型关系。

注意到这个定义并没有把上面的 μ-folding 规则准确包含进来：为使其可逆，我们在最后一条子句和倒数第二条子句之间引入了反对称式（否则，子句将互相重叠）。然而，正如在下一个练习中将会看到，S_m 产生的子类型关系与更本质的（完全对应于）推导规则的产生函数[①] S_d 相同。

21.8.5 练习[★★★]：写出刚才提到的函数 S_d，并证明它是不可逆的。证明 $\nu S_d = \nu S_m$。

产生函数 S_m 是不可逆的，因为与之对应的支持函数为良定义形式：

$$support_{S_m}(S,T) = \begin{cases} \varnothing & \text{if } T = Top \\ \{(S_1,T_1),(S_2,T_2)\} & \text{if } S = S_1 \times S_2 \text{ and} \\ & \quad T = T_1 \times T_2 \\ \{(T_1,S_1),(S_2,T_2)\} & \text{if } S = S_1 \to S_2 \text{ and} \\ & \quad T = T_1 \to T_2 \\ \{(S,[X \mapsto \mu X.T_1]T_1)\} & \text{if } T = \mu X.T_1 \\ \{([X \mapsto \mu X.S_1]S_1,T)\} & \text{if } S = \mu X.S_1 \text{ and} \\ & \quad T \neq \mu X.T_1, T \neq Top \\ \uparrow & \text{其他} \end{cases}$$

到目前为止，μ 类型上的子类型关系是独立于（前面定义的）树类型上子类型关系进行介绍的。因为我们考虑 μ 类型时是将其表示为有限形式的常规类型，所以还有必要确保这两种子类型概念互相对应。下一定理(21.8.7)就建立这种对应。但是，首先需要引入一个引理。

21.8.6 引理：假设 $R \subseteq \mathcal{T}_m \times \mathcal{T}_m$ 是 S_m 一致的。对任意 $(S,T) \in R$，存在某 $(S',T') \in R$ 使得 $treeof(S',T') = treeof(S,T)$ 且 S′和 T′都不以 μ 开头。

证明：归纳 S 和 T 前端的 μ 总数。如果 S 和 T 都不以 μ 开头，那么 $(S',T') = (S,T)$。另一方面，如果 $(S,T) = (S,\mu X.T_1)$，那么根据 R 的 S_m 一致性，有 $(S,T) \in S_m(R)$，所以 $(S'',T'') = (S,[X \mapsto \mu X.T_1]T_1) \in R$。因为 T 是收缩的，展开 T 的结果 T″前端的 μ 比 T 的更少。根据归纳假设，可知存在某 $(S',T') \in R$ 使得 S′和 T′都不以 μ 开头，且 $treeof(S'',T'') = (S',T')$。因为根据 $treeof$ 的定义，$treeof(S,T) = treeof(S'',T'')$，序对 (S',T') 就是我们所需要的。出现 $(S,T) = (\mu X.S_1,T)$ 的情况类似。

21.8.7 定理：设 $(S,T) \in \mathcal{T}_m \times \mathcal{T}_m$。当且仅当 $treeof(S,T) \in \nu S$ 时，有 $(S,T) \in \nu S_m$。

证明：首先，考虑"仅当"的情况，即 $(S,T) \in \nu S_m$ 推出 $treeof(S,T) \in \nu S$ 的情况。设 $(A,B) = treeof(S,T) \in \mathcal{T} \times \mathcal{T}$，根据共归纳原理，如果可以找到一个 S 一致集 $Q \in \mathcal{T} \times \mathcal{T}$ 使得 $(A,B) \in Q$，则可得出结果。这里认为 $Q = treeof(\nu S_m)$ 就是这样的集合。为了证明这一点，必须证明对每个 $(A',B') \in Q$ 都有 $(A',B') \in S(Q)$。

设 $(S',T') \in \nu S_m$ 为 μ 类型序对，满足 $treeof(S',T') = (A',B')$。根据引理(21.8.6)，先假设 S′和 T′都不以 μ 开头。因为 νS_m 是 S_m 一致的，所以 (S',T') 必须由定义 S_m 其中的某条语句支持，即它必须具备以下某一特性：

情况： $(S',T') = (S',Top)$

① S_d 中的 d 是用来特别强调该函数是基于 μ-folding 的"声明性"推导规则，以区分于 S_m 中采用的"算法"规则。

那么根据 S 的定义有 $B' = Top$ 且 $(A', B') \in S(Q)$。

情况： $(S', T') = (S_1 \times S_2, T_1 \times T_2)$，　其中 $(S_1, T_1), (S_2, T_2) \in \nu S_m$

根据 *treeof* 定义，有 $B' = treeof(T') = B_1 \times B_2$，其中 $B_i = treeof(T_i)$。同理，有 $A' = A_1 \times A_2$，其中 $A_i = treeof(S_i)$。将 *treeof* 应用于这些序对，有 $(A_1, B_1), (A_2, B_2) \in Q$。但是，根据 S 的定义，可知 $(A, B) = (A_1 \times A_2, B_1 \times B_2) \in S(Q)$。

情况： $(S', T') = (S_1 \to S_2, T_1 \to T_2)$，　其中 $(T_1, S_1), (S_2, T_2) \in \nu S_m$

同理。

接下来，考虑"当"的情况，即证明 $treeof(S,T) \in \nu S$ 推导出 $(S,T) \in \nu S_m$。根据共归纳原理，需要找到一个 S_m 一致集 $R \in \mathcal{T}_m \times \mathcal{T}_m$ 使 $(S,T) \in R$。这里认为 $R = \{(S', T') \in \mathcal{T}_m \times \mathcal{T}_m \mid treeof(S', T') \in \nu S\}$ 为这样的集合。显然，$(S,T) \in R$。为了完成证明，必须先说明 $(S', T') \in R$ 能推导出 $(S', T') \in S_m(R)$。

注意到，因为 νS 是 S 一致的，所以任何序对 $(A', B') \in \nu S$ 必须具有 (A', Top)，$(A_1 \times A_2, B_1 \times B_2)$，$(A_1 \to A_2, B_1 \to B_2)$ 其中一种形式。由此，并根据 *treeof* 的定义，可知任意序对 $(S', T') \in R$ 必须具有 (S', Top)，$(S_1 \times S_2, T_1 \times T_2)$，$(S_1 \to S_2, T_1 \to T_2)$，$(S', \mu X.T_1)$ 或 $(\mu X.S_1, T')$ 其中一种形式。我们依次考虑其中的每种情况：

情况： $(S', T') = (S', Top)$

那么由 S_m 的定义，立即可得出 $(S', T') \in S_m(R)$。

情况： $(S', T') = (S_1 \times S_2, T_1 \times T_2)$

设 $(A', B') = treeof(S', T')$，那么有 $(A', B') = (A_1 \times A_2, B_1 \times B_2)$，其中 $A_i = treeof(S_i)$，$B_i = treeof(T_i)$。因为 $(A', B') \in \nu S$，νS 的 S 一致性可得出 $(A_i, B_i) \in \nu S$，这样根据 R 的定义，可得出 $(S_i, T_i) \in R$。S_m 的定义产生 $(S', T') = (S_1 \times S_2, T_1 \times T_2) \in S_m(R)$。

情况： $(S', T') = (S_1 \to S_2, T_1 \to T_2)$

同理。

情况： $(S', T') = (S', \mu X.T_1)$

设 $T'' = [X \mapsto \mu X.T_1]T_1$。根据定义有 $treeof(T'') = treeof(T')$，这样由 R 的定义，有 $(S', T'') \in R$ 且由 S_m 的定义有 $(S', T') \in S_m(R)$。

情况： $(S', T') = (\mu X.S_1, T')$

如果 $T' = Top$ 或 T' 以 μ 开头，那么上面任何一种情况可成立；否则，参数就同前面一个相同。

考虑到无穷树类型(用有限 μ 表达式来表示)之间一般的子类型关系，本定理所建立的 μ 类型(本节所定义的)上子类型间的对应是可靠且完备的。

21.9　计算子表达式

用 μ 类型[参见(21.8.4)]上的子类型关系的特殊支持函数 $support_{S_m}$ 来实例化一般算法 gfp^t[参见(21.6.4)]会产生如图 21.4 所示的子类型算法。21.6 节说明了如果对任意 μ 类型序

对(S,T), $reachable_{S_m}$(S,T)都是有限的,那么该算法一定会终止。本节将证明确实是这样的[参见命题(21.9.11)]。

乍看一眼,该性质好像是显而易见成立的,但证明起来却要付出惊人的工作量。问题在于存在两种定义 μ 类型的"子表达式闭集"的方式。一种称为自顶向下子表达式,直接对应于 $support_{S_m}$ 产生的子表达式;另一种称为自底向上子表达式,直接可用来证明每个封闭 μ 类型的子表达式闭集都是有限的。要证明可终止性,先要对应出这两个集合,然后说明前者是后者的子集[参见命题(21.9.10)]。本节的讨论是依据 Brandt 和 Henglein 的论著(1997)。

```
subtype(A, S, T)  =  if (S, T) ∈ A, then
                       A
                     else let A0 = A ∪ {(S, T)} in
                       if T = Top, then
                         A0
                       else if S = S1 × S2 and T = T1 × T2, then
                         let A1 = subtype(A0, S1, T1) in
                         subtype(A1, S2, T2)
                       else if S = S1→S2 and T = T1→T2, then
                         let A1 = subtype(A0, T1, S1) in
                         subtype(A1, S2, T2)
                       else if T = μX.T1, then
                         subtype(A0, S, [X ↦ μX.T1]T1)
                       else if S = μX.S1, then
                         subtype(A0, [X ↦ μX.S1]S1, T)
                       else
                         fail
```

图 21.4 μ 类型的子类型化算法

21.9.1 定义:对 μ 类型 S 和 T,如果序对(S,T)是产生函数:

$$
\begin{aligned}
TD(R) \quad = \quad & \{(T,T) \mid T \in \mathcal{T}_m\} \\
\cup \quad & \{(S, T_1 \times T_2) \mid (S, T_1) \in R\} \\
\cup \quad & \{(S, T_1 \times T_2) \mid (S, T_2) \in R\} \\
\cup \quad & \{(S, T_1 \to T_2) \mid (S, T_1) \in R\} \\
\cup \quad & \{(S, T_1 \to T_2) \mid (S, T_2) \in R\} \\
\cup \quad & \{(S, \mu X.T) \mid (S, [X \mapsto \mu X.T]T) \in R\}
\end{aligned}
$$

的最小不动点,则 S 为 T 自顶向下的子表达式,记为 S ⊑ T。

21.9.2 练习[★]:将关系 S ⊑ T 视为推导规则的集合,给出等价定义。

从 $support_{S_m}$ 的定义容易看出,对任何 μ 类型 S 和 T,所有在 $support_{S_m}$(S,T)中的序对都是由自顶向下的 S 和 T 的子表达式组成。

21.9.3 引理:如果(S′,T′)∈ $support_{S_m}$(S,T),那么 S′和 T′满足:S′⊑ S 或 S′⊑ T;T′⊑ S 或 T′⊑ T。

证明:直接对 $support_{S_m}$ 的定义进行观察即可得证。

还有,自顶向下的子表达式关系也有传递性。

21.9.4　引理: 如果 $S \sqsubseteq U$ 且 $U \sqsubseteq T$,那么 $S \sqsubseteq T$。

证明: 引理的表述等价于对 $\forall U$,有 $T.U \sqsubseteq T \Rightarrow (\forall S.S \sqsubseteq U \Rightarrow S \sqsubseteq T)$。换句话说,必须说明 $\mu(TD) \sqsubseteq R$,其中 $R = \{(U,T) \mid \forall S.S \sqsubseteq U \Rightarrow S \sqsubseteq T\}$。根据归纳原则,有 R 是 TD 封闭的,即 $TD(R) \subseteq R$。所以假设 $(U,T) \in TD(R)$,现在对 TD 定义中子句出现的几种情况来讨论:

情况:　$(U,T) = (T,T)$

显然,　$(T,T) \in R$

情况:　$(U,T) = (U,T_1 \times T_2)$ and $(U,T_1) \in R$

因为 $(U,T_1) \in R$,所以对所有的 S 必有 $S \sqsubseteq U \Rightarrow S \sqsubseteq T_1$。根据 $\sqsubseteq$ 的定义,对所有的 S 必有 $S \sqsubseteq U \Rightarrow S \sqsubseteq T_1 \times T_2$。这样,根据 R 的定义,有 $(U,T) = (U,T_1 \times T_2) \in R$。其他情况类似。

若将前两个引理合起来考虑,可提出关于自顶向下的子表达式命题:

21.9.5　命题: 如果 $(S',T') \in reachable_{S_m}(S,T)$,那么 $S' \sqsubseteq S$ 或 $S' \sqsubseteq T$,及 $T' \sqsubseteq S$ 或 $T' \sqsubseteq T$。

证明: 利用 $\sqsubseteq$ 的传递性,对 $reachable_{S_m}$ 的定义直接归纳得证。

根据上面的命题及任意 μ 类型 U 都有有限数量的自顶向下的子表达式这一事实,可得出 $reachable_{S_m}(S,T)$ 的有限性[参见后面的命题(21.9.11)]。但任意 μ 类型 U 都有有限数量的自顶向下的子表达式这一点不容易从 μ 的定义看出,用 TD 的定义对 U 进行结构化归纳没什么效果,因为 TD 的最后子句破坏了归纳:构造 $U = \mu X.T$ 的子表达式时,它得出的是更大的表达式 $[X \mapsto \mu X.T]T$。

另一个概念:自底向上子表达式,只要在计算子表达式之后(而不是之前)执行递归变量的 μ 类型代换,就可以避免出现这个问题。这样能使有限性证明更简单。

21.9.6　定义: 对 μ 类型 S 和 T,如果序对 (S,T) 在如下产生函数:

$$
\begin{aligned}
BU(R) \;=\; & \{(T,T) \mid T \in \mathcal{T}_m\} \\
\cup\; & \{(S,T_1 \times T_2) \mid (S,T_1) \in R\} \\
\cup\; & \{(S,T_1 \times T_2) \mid (S,T_2) \in R\} \\
\cup\; & \{(S,T_1 \rightarrow T_2) \mid (S,T_1) \in R\} \\
\cup\; & \{(S,T_1 \rightarrow T_2) \mid (S,T_2) \in R\} \\
\cup\; & \{([X \mapsto \mu X.T]S, \mu X.T) \mid (S,T) \in R\}
\end{aligned}
$$

的最小不动点中,则 S 是 T 的自底向上子表达式,记为 $S \preccurlyeq T$。

该子表达式的新定义与老的不同之处仅在于子句中类型的开头有 μ 绑定器。为获得该类型的自顶向下子表达式,要先把它展开,然后再将展开的子表达式放在一起。而为获得自底向上的子表达式,要先将函数体中的子表达式(不一定为封闭的)收集一起,然后应用展开代换封闭它们。

21.9.7　练习[★★]: 若将关系 $S \preccurlyeq T$ 看成是推导规则的集合,给出等价的定义。

很容易证明一个表达式只有有限多个自底向上的子表达式。

21.9.8 引理:对每个 T,$\{S|\ S\leqslant T\}$是有限的。

证明:根据 *BU* 和$\leqslant$的定义,可得出下面的这些结论:

- if $T = Top$ or $T = X$ then $\{S|S\leqslant T\} = \{T\}$;
- if $T = T_1 \times T_2$ or $T = T_1 \to T_2$ then $\{S|S\leqslant T\} = \{T\}\cup(\{S|S\leqslant T_1\}\cup\{S|S\leqslant T_2\}$;
- if $T = \mu X.T'$ then $\{S|S\leqslant T\} = \{T\}\cup\{[X\mapsto T]S|S\leqslant T'\}$。

根据这些结论可对 T 直接进行结构化归纳即可得证。

为证明类型的自底向上子表达式包括了自顶向下的子表达式,还需要一个将自底向上的子表达式与代换联系起来的引理。

21.9.9 引理:如果 $S\leqslant[X\mapsto Q]T$,那么对满足 $S'\leqslant T$ 的 S',有 $S\leqslant Q$ 成立或 $S=[X\mapsto Q]S'$成立。

证明:对 T 进行结构化归纳。

情况: $T = Top$

只有 *BU* 的自反子句允许 Top 为序对的右端元素,所以必须让 $S = Top$。若设 $S' = Top$ 会得到想要的结果。

情况: $T = Y$

如果 $Y = X$,有 $S\leqslant[X\mapsto Q]T = Q$,则根据这个假设可以得到理想的结果。如果 $Y\neq X$,有 $S=[X\mapsto Q]T = Y$。只有 BU 的自反子句能判定该序对,所以必须使 $S = Y$。令 $S' = Y$ 来得出理想的结果。

情况: $T = T_1\times T_2$

有 $S\leqslant[X\mapsto Q]T = [X\mapsto Q]T_1\times[X\mapsto Q]T_2$。根据 *BU* 的定义,可知存在三种使 S 成为这种乘积类型的自底向上的子表达式的方法。下面逐一来考虑:

子情况: $S = [X\mapsto Q]T$

那么可使 $S' = T$。

子情况: $S\leqslant[X\mapsto Q]T_1$

根据归纳假设,对某 $S'\leqslant T$ 有 $S\leqslant Q$(这种情况已经讨论了)或 $S=[X\mapsto Q]S'$。根据 *BU* 的定义,后者能得出理想的结果 $S'\leqslant T_1\times T_2$。

子情况: $S\leqslant[X\mapsto Q]T_2$

同理。

情况: $T = T_1\to T_2$

类似于乘积类型情况。

情况: $T = \mu Y.T'$

有 $S\leqslant[X\mapsto Q]T = \mu Y.[X\mapsto Q]T'$。有两种使 S 成为该 μ 类型的自底向上子表达式的方法:

子情况: $S = [X\mapsto Q]T$

让 $S' = T$

子情况: $S = [Y\mapsto \mu Y.[X\mapsto Q]T']S_1$ 其中 $S_1\leqslant[X\mapsto Q]T'$

应用归纳假设可得出两种可能结果：

- $S_1 \leqslant Q$。根据对囿变量名称的约定，知道 $Y \notin FV(Q)$，所以必有 $Y \notin FV(S_1)$。但 $S = [Y \mapsto \mu Y.[X \mapsto Q]T']S_1 = S_1$，所以 $S \leqslant Q$。
- 对某 S_2 有 $S_1 = [X \mapsto Q]S_2$ 使得 $S_2 \leqslant T'$。这种情况下，$S = [Y \mapsto \mu Y.[X \mapsto Q]T']S_1 = [Y \mapsto \mu Y.[X \mapsto Q]T'][X \mapsto Q]S_2 = [X \mapsto Q][Y \mapsto \mu Y.T']S_2$。令 $S' = [Y \mapsto \mu Y.S']S_2$ 来得到理想的结果。

证明的最后一条说明了 μ 类型的每个自顶向下子表达式都可以从它的自底向上的子表达式中找到。

21.9.10 命题：如果 $S \sqsubseteq T$，那么 $S \leqslant T$。

证明：一开始想说明 $\mu TD \subseteq \mu BU$。根据归纳原则，如果能说明 μBU 是 TD 封闭的，即 $TD(\mu BU) \subseteq \mu BU$，就可以得出 $\mu TD \subseteq \mu BU$。另外，还想说明 $(A,B) \in TD(\mu BU)$ 能推导出 $(A,B) \in \mu BU = BU(\mu BU)$。如果每个可以从 μBU 中产生 (A,B) 的 TD 子句能与 μBD 中产生 (A,B) 的 BU 子句相匹配，那么第二种想说明的即为真。在 TD 的定义中，除了最后一条子句，其余的子句想使之为真比较勉强，因为它们与对应的 BU 子句完全相同。对最后一条子句，有 $(A,B) = (S, \mu X.T) \in TD(\mu BU)$ 及 $(S, [X \mapsto \mu X.T]T) \in \mu BU$，或等价地，$S \leqslant [X \mapsto \mu X.T]T$。根据引理(21.9.9)，有 $S \leqslant \mu X.T$，即 $(S, \mu X.T) \in \mu BU$（这是需要的），或者对满足 $(S', T) \in \mu BU$ 的某 S' 有 $S = [X \mapsto \mu X.T]S'$。根据 BU 的最后子句，后者可推导出 $(S, \mu X.T) \in BU(\mu BU) = \mu BU$。

21.9.11 命题：对任意 μ 类型 S 和 T，集合 $reachable_{S_m}(S,T)$ 是有限的。

证明：对 S 和 T，设 Td 为它们所有的自顶向下的子表达式集合，Bu 为它们所有自底向上的子表达式集合。根据命题(21.9.5)，有 $reachable_{S_m}(S,T) \subseteq Td \times Td$。根据命题(21.9.10)，有 $Td \times Td \subseteq Bu \times Bu$。根据引理(21.9.8)，可知后者是有限集。所以可推导出 $reachable_{S_m}(S,T)$ 为有限的。

21.10 关于指数级算法的闲话

21.9 节开始提到的子类型算法（参见图 21.4）可通过让它返回布尔值来进一步简化，而不是采取一个新的假设集（参见图 21.5）。中间结果 $subtype^{ac}$ 对应于 Amadio 和 Cardelli 的检查子类型化的算法(1993)。它计算的关系与子类型计算的关系相同，但效率要低些，因为它不能记住发生在→和 × 情况的递归调用的子类型关系中的类型序对。这看似无关紧要的改变会导致算法中递归调用数量的激增。但是子类型进行的递归调用数量是与两个参数类型中的子表达式总数的平方成比例［可通过检查引理(21.9.8)和命题(21.9.11)中的证明了解这一点］，发生 $subtype^{ac}$ 时，它将成指数级增长。

$$
\begin{array}{rcl}
subtype^{ac}(A,S,T) & = & \text{if } (S,T) \in A, \text{ then } \mathit{true} \\
& & \text{else let } A_0 = A \cup (S,T) \text{ in} \\
& & \quad \text{if } T = Top, \text{ then } \mathit{true} \\
& & \quad \text{else if } S = S_1 \times S_2 \text{ and } T = T_1 \times T_2, \text{ then} \\
& & \qquad subtype^{ac}(A_0, S_1, T_1) \text{ and} \\
& & \qquad subtype^{ac}(A_0, S_2, T_2) \\
& & \quad \text{else if } S = S_1 \to S_2 \text{ and } T = T_1 \to T_2, \text{ then} \\
& & \qquad subtype^{ac}(A_0, T_1, S_1) \text{ and} \\
& & \qquad subtype^{ac}(A_0, S_2, T_2) \\
& & \quad \text{else if } S = \mu X.S_1, \text{ then} \\
& & \qquad subtype^{ac}(A_0, [X \mapsto \mu X.S_1]S_1, T) \\
& & \quad \text{else if } T = \mu X.T_1, \text{ then} \\
& & \qquad subtype^{ac}(A_0, S, [X \mapsto \mu X.T_1]T_1) \\
& & \quad \text{else } \mathit{false}.
\end{array}
$$

图 21.5 Amadio 和 Cardelli 的子类型化算法

$subtype^{ac}$的指数行为可在下例中清楚地体现出来。将类型簇 S_n 和 T_n 归纳定义为：

$$
\begin{array}{lcllcl}
S_0 & = & \mu X.Top \times X & S_{n+1} & = & \mu X.X \to S_n \\
T_0 & = & \mu X.Top \times (Top \times X) & T_{n+1} & = & \mu X.X \to T_n.
\end{array}
$$

因为 S_n 和 T_n 都分别出现一次 S_{n-1} 和 T_{n-1}，所以它们的长度(将简写展开后)随 n 成线性增长。然而，检查 $S_n <: T_n$ 会产生指数级推导，这一点可从下面的递归调用产生的结果看出：

$$
\begin{array}{rl}
& subtype^{ac}(\varnothing, S_n, T_n) \\
= & subtype^{ac}(A_1, S_n \to S_{n-1}, T_n) \\
= & subtype^{ac}(A_2, S_n \to S_{n-1}, T_n \to T_{n-1}) \\
= & subtype^{ac}(A_3, T_n, S_n) \text{ and } \underline{subtype^{ac}(A_3, S_{n-1}, T_{n-1})} \\
= & subtype^{ac}(A_4, T_n \to T_{n-1}, S_n) \text{ and} \ldots \\
= & subtype^{ac}(A_5, T_n \to T_{n-1}, S_n \to S_{n-1}) \text{ and} \ldots \\
= & subtype^{ac}(A_6, S_n, T_n) \text{ and } \underline{subtype^{ac}(A_6, T_{n-1}, S_{n-1})} \text{ and} \ldots \\
= & \text{etc..}
\end{array}
$$

其中：

$$
\begin{array}{lcl}
A_1 & = & \{(S_n, T_n)\} \\
A_2 & = & A_1 \cup \{(S_n \to S_{n-1}, T_n)\} \\
A_3 & = & A_2 \cup \{(S_n \to S_{n-1}, T_n \to T_{n-1})\} \\
A_4 & = & A_3 \cup \{(T_n, S_n)\} \\
A_5 & = & A_4 \cup \{(T_n \to T_{n-1}, S_n)\} \\
A_6 & = & A_5 \cup \{(T_n \to T_{n-1}, S_n \to S_{n-1})\}.
\end{array}
$$

注意到初始调用 $subtype^{ac}(\varnothing, S_n, T_n)$的结果是两个含 S_{n-1} 和 T_{n-1} 的相同形式的递归调用(用下划线表示)。而它们又会产生两个含 S_{n-2} 和 T_{n-2} 的递归调用，依次类推。按这样的比例计算下来总的递归调用数为 2^n。

21.11 子类型化同构递归类型

在 20.2 节中，曾说过一些递归类型采用了同构递归表示方式，它将递归类型的折叠和展

开已明确地注明了项构造子 fold 和 unfold。在这种语言中，μ 类型构造子是"严格的"，因为它在类型中的位置会影响该类型项的使用方式。

如果在同构递归类型的语言中添加子类型，μ 构造子的严格性还会影响子类型关系。本章中采用的大部分方法都是"除去限制，再子类型化"，但这里不采用这种办法，而是直接在递归类型中定义子类型规则。

最常见的同构递归子类型化定义为 Amber 规则，因为它是由 Cardelli 的 Amber 语言(1986)广泛使用的：

$$\frac{\Sigma, X<:Y \vdash S <: T}{\Sigma \vdash \mu X.S <: \mu Y.T} \qquad \text{(S-Amber)}$$

直观地看，该规则可读为"在某假设集 Σ 下，若加上一个条件 $X <: Y$ 有 $S <: T$，则 $\mu X.S$ 为 $\mu Y.T$ 的子类型"[①]。这里的 Σ 是递归变量序对集合，包括了所有经过判断的递归类型序对。这种假定可以用另一个子类型规则来表示：

$$\frac{(X <: Y) \in \Sigma}{\Sigma \vdash X <: Y} \qquad \text{(S-Assumption)}$$

即，如果现在假设 $X <: Y$，则可以将其加入。

将这两个规则加入到第 16 章提出的子类型化算法(并扩展别的规则以从前提将 Σ 传递给结论)，这样会产生一个行为类似于图 21.5 中的 $subtype^{ac}$ 算法的新算法，其中 Σ 起到了 A 的作用。不同之处在于：(1)只有当<:的两边同时出现了递归类型时立即将它们展开；(2)对递归类型不做代换(将其视为变量保留)，这样容易看出算法终止。

在规范化类型系统中(如第19章的轻量级Java)出现的子类型规则与Amber规则密切相关。

21.11.1　练习[推荐，★★]：请利用相等递归定义(但不能用 Amber 规则)来找到递归类型 S 和 T，使得 $S <: T$。

21.12　注释

本章以 Gapeyev，Levin 和 Pierce(2000)所著的手册为基础。

想了解有关共归纳的背景知识可参考 Barwise 和 Moss 的"Vicious Circles"(1996)，Gordon 著的《共归纳和函数编程手册》(1995)，及 Milner 和 Tofte 的程序语言语义中关于共归纳声明性文章(1991a)。想了解关于单调函数和不动点的基础知识，可参考 Aczel(1977)和 Davey 以及 Priestley(1990)。

计算机科学中共归纳证明方法的使用要追溯到 20 世纪 70 年代，例如 Milner(1980)和 Park(1981)所著的关于并发的文章；也可参见 Arbib 和 Manes 所著的自动控制理论中对二元性的范畴性讨论(1975)。但是归纳前带上了"co-"形式早已为数学家们所熟悉，比如，它在通用代数和范畴论中有明确的论述。在 Aczel 的关于非良定集合的开创性书籍(1988)中简单介绍了其发展史。

---

① 注意该规则，与其他大部分两边含绑定结构的规则(如图 26.1 中的 S-All)不同，它要求围变量 X 和 Y 在规则应用之前重新命名为不同的名称。

Amadio 和 Cardelli(1993)给出了第一个关于递归类型的子类型化算法。他们在论文中定义了 3 种类型关系:(1)无穷树间的包含关系;(2)检查 μ 类型之间子类型关系的算法;(3)定义为声明性推导规则集的最小不动点的 μ 类型之间引用子类型关系。这些关系都证明为等价的,并与基于部分等价关系的模型结构有关。不采用共归纳,出于无穷树的考虑,引入了一个无穷树的有限逼近概念,这个概念在许多证明中都起到重要的作用。

Brandt 和 Henglein(1997)根据 Amadio 和 Cardelli 的结果引入了一个新的使子类型关系更加可靠和完备的归纳性公理化形式,从而揭示了 Amadio 和 Cardelli 系统的共归纳本质。公理化中箭头(固定)规则都进一步具体化了系统的共归纳性。这篇文章描述了推出关系(由共归纳定义)中归纳性公理化的一般方法,并对子类型算法可终止性进行了证明。21.9 节与后者的证明密切相关。Brandt 和 Henglein 还发现了算法的复杂度为 $O(n^2)$。

Kozen, Palsberg 和 Schwartzbach(1993)通过研究发现正则递归类型对应于有状态记录的自动机,从而得出了一个更巧妙的二次子类型化算法。他们定义了两个自动机的乘积,产生一个约定词操作,即当且仅当最初自动机的类型不是子类型关系时接收一个词。线性时间空集测试可以接近子类型的问题。它再加上乘积结构的二次复杂度及类型到自动机的线性时间转换,就能得出整个二次时间的复杂度。

Hosoya, Vouillon 和 Pierce(2001)用与自动机理论有关的方法,将递归类型(带联合类型)与调整为 XML 处理过程的子类型化算法中的树自动机联系起来。

Jim 和 Palsberg(1999)针对子类型和递归类型语言提出了类型重构概念(参见第 22 章)。如本章所做,他们用共归纳的观点来看待无穷树的子类型关系,并提出将子类型检查算法作为一个在给定的类型序对中建立最小模拟的(即在术语学上称为一致集)过程。他们定义类型上的一致性和 $P1$ 闭包的概念,这正好与本书的一致性和可达集合对应。

如果你思考了足够长的时间,那么会明白这是显然成立的。

——Saul Gorn

第五部分　多　　态

第22章　类型重构[1]

到目前为止,我们所见到的对演算进行的类型检查算法都有明确的类型注释——尤其是要求注释 lambda 抽象的参数类型。在本章中,提出了一个更有用的类型重构算法,该算法能够为一个项计算主类型,该项只进行了部分注释或全未注释。相关的算法还处于某些语言,如 ML 和 Haskell 等的核心地位。

将类型重构与其他语言的特征结合起来是一件难事。尤其在记录型和子类型上有很大的困难。为了使问题简单化,这里只考虑对简单类型进行类型重构;从 22.8 节开始将对其他的组合方式进一步讨论。

22.1　类型变量和代换

在前面章节的一些演算中,我们已经假设类型集合包括了无穷的没有解释的基类(参见 11.1 节)。不同于有解释的基类如 Bool 和 Nat,这些类型没有引入或消去项的操作;直觉上说,它们只是一些特殊类型的符号,而实际的含义我们无需领会。在本章中,将会不停地提出一些问题,比如"如果给项 t 中的符号 X 一个具体的类型 Bool,是否可获得一个可类型化的项?"。换句话说,我们将把没有解释的基类型当做类型变量,用其他类型来代换或将其实例化。

本章为采用技术手段说明问题,将对类型变量的代换操作分成两部分:描述一个从类型变量到类型的映射 σ(称为类型代换),再将该映射应用到一个特殊类型 T 上以获得实例 σT。例如,定义 σ = [X ↦ Bool],然后将 σ 应用于类型 X→X 得到 σ(X→X) = Bool→Bool。

22.1.1　定义: 通常,类型代换(当已知讨论的是类型时可直接简称为代换)指一个从类型变量转换到类型的有限映射。例如,[X ↦ T, Y ↦ U]作为将 X 与 T, Y 与 U 联合起来的类型代换。用 *dom*(σ)来表示出现在 σ 的序对中左端的类型变量集合,用 *range*(σ)表示出现在右端的类型集合。注意:相同的变量可能同时出现在代换定义域和值域的位置上。类似于项代换,如果出现了上述的情况时,表示所有的代换都是同时进行的;例如,[X ↦ Bool, Y ↦ X→X]是将 X 映射成 Bool,将 Y 映射成 X→X,而不是 Bool→Bool。

代换可以用下面的方式来表示:

$$
\begin{array}{lcl}
\sigma(\mathtt{X}) & = & \begin{cases} \mathtt{T} & \text{如果 } (\mathtt{X} \mapsto \mathtt{T}) \in \sigma \\ \mathtt{X} & \text{如果 X 不是 } \sigma \text{ 的定义域} \end{cases} \\
\sigma(\mathtt{Nat}) & = & \mathtt{Nat} \\
\sigma(\mathtt{Bool}) & = & \mathtt{Bool} \\
\sigma(\mathtt{T_1} \to \mathtt{T_2}) & = & \sigma\mathtt{T_1} \to \sigma\mathtt{T_2}
\end{array}
$$

---

[1] 本章学习的系统是包括布尔型、数型及无穷基类集合的简单类型的 lambda 演算(参见图 9.1)。对应的 OCaml 实现是 recon 和 fullrecon。

注意,在类型代换过程中,不需采取任何准备措施来避免变量捕获,因为在类型表达语言中没有什么类型表达结构绑定类型变量(将在第 23 章谈到这个问题)。

类型代换可以逐点扩充至上下文中,定义如下:

$$\sigma(x_1:T_1,\ldots,x_n:T_n) = (x_1:\sigma T_1,\ldots,x_n:\sigma T_n).$$

类似地,如果一个代换应用于出现在项 t 注释中的所有类型,则该代换应用于该项。

如果 σ 和 γ 都是代换,可以用 $\sigma\circ\gamma$ 来组成新的代换形式:

$$\sigma\circ\gamma = \left[\begin{array}{ll} X\mapsto\sigma(T) & \text{其中 } (X\mapsto T)\in\gamma \\ X\mapsto T & \text{其中 } (X\mapsto T)\in\sigma \text{ 且 } X\notin dom(\gamma) \end{array}\right]$$

注:$(\sigma\circ\gamma)S=\sigma(\gamma S)$。

类型代换的一个重要特性是它们保留了类型声明的有效性:如果包含变量的项是良类型的,则所有它的代换实例都是良类型的。

22.1.2 定理[类型代换下的类型化保持]:设 σ 是任意类型代换,且 $\Gamma\vdash t:T$,则 $\sigma\Gamma\vdash\sigma t:\sigma T$。

证明:可直接从类型化推导中归纳得出。

22.2 类型变量的两个观点

设 t 是一个包含类型变量的项,Γ 是一个相关的上下文(可能也含有类型变量)。我们可以就 t 提出两个截然不同的问题:

1. “t 是否在所有代换实例中都是良类型?”,也就是问,对一切 σ 都存在 T 满足 $\sigma\Gamma\vdash\sigma t:T$?
2. “t 是否存在为良类型的代换实例?”,即能否找到一个 σ 存在 T 满足 $\sigma\Gamma\vdash\sigma t:T$?

对第一个问题,在类型检查过程中类型变量应始终保持抽象,这样才能确保一个良类型的项无论在后来的类型代换中用什么样的具体类型代换变量,都能正常使用。例如,对项:

```
λf:X→X. λa:X. f (f a);
```

有类型(X→X)→X→X,无论什么时候用一个具体的类型 T 来代替 X 时,实例:

```
λf:T→T. λa:T. f (f a);
```

都是良类型。保持这种类型变量抽象可引出参数化多态,也就是类型变量的使用使得一个项在具体的上下文中具有不同的类型。在 22.7 节中,还会谈到参数化多态的问题,并在第 23 章中更深入地加以讨论。

对第二个问题,原始项 t 甚至可能不是良类型的,而我们所关心的问题是能否通过它的类型变量选择合适的值来将该项实例化为良类型。比如,有项:

```
λf:Y. λa:X. f (f a);
```

如它是不可类型化的,但如果用 Nat→Nat 代替 Y,用 Nat 代替 X,就可获得:

```
λf:Nat→Nat. λa:Nat. f (f a);
```

可得到类型(Nat→Nat)→Nat→Nat。或者简单一点,用 X→X 来代替 Y,也能得到良类型的项:

```
λf:X→X. λa:X. f (f a);
```

尽管它仍含变量。确实,该项是 λf:Y.λa:X.f(f a)最一般性的实例,其意义就在于它使用最简单的类型变量值就能产生一个良类型的项。

在寻找类型变量有效实例的过程中,出现了一个新概念类型重构(有时也称类型推断),它意味着由编译器来帮助添加那些编程人员忽略掉的类型信息。受条件限制,如在 ML 语言中,我们允许编程人员忽略所有的类型注释,仅依照原始的无类型的 lambda 演算中的语法写程序。在做分析时,对每个原始的 lambda 抽象 λx.t 用类型变量进行注释,写成 λx:X.t,其中选择的 X 与程序中其他抽象使用的类型变量都不同。然后采取类型重构来找到一个使项通过类型检查的最一般化值(这一步在出现了 ML 的 let 多态后变得更加复杂了,在 22.6 节和 22.7 节中还会谈到)。

为形式化类型重构,需要一个简明的方式来讨论,在项及其相关的上下文中,如何用类型来代换类型变量以得到有效的类型化语句[①]。

22.2.1 定义: 设 Γ 为上下文,t 为项,(Γ,t)的解是指一个序对(σ,T),使得 $\sigma\Gamma \vdash \sigma t:T$ 成立。

22.2.2 实例: 设 Γ = f:X,a:Y 且 t = f a,那么:

```
([X ↦ Y→Nat], Nat)             ([X ↦ Y→Z], Z)
([X ↦ Y→Z, Z ↦ Nat], Z)        ([X ↦ Y→Nat→Nat], Nat→Nat)
([X ↦ Nat→Nat, Y ↦ Nat], Nat)
```

都是(Γ,t)的解。

22.2.3 练习[★ ↛]: 在空的上下文中,为下列项找出三个不同的解:

```
λx:X. λy:Y. λz:Z. (x z) (y z).
```

22.3 基于约束的类型化

对给定的项 t 和上下文 Γ,我们提出一个算法,用来计算一个约束集——类型表达式(可包含类型变量)之间的等式,该集合对任何(Γ,t)的解都满足。该算法的想法与通常的类型检查算法概念基本上是相同的,惟一的区别是它不是用来检查约束,而是把它们记录下来以备以后考虑。例如,对应用 $t_1 t_2$ 有 $\Gamma \vdash t_1:T_1$ 和 $\Gamma \vdash t_2:T_2$,不是检查 T_1 有形式 $T_2 \to T_{12}$ 并将 T_{12} 作为应用的类型返回,而是选择一个新的类型变量 X,记录约束式 $T_1 = T_2 \to X$,将 X 作为应用的类型返回。

22.3.1 定义: 约束集 C 指一个等式集 $\{S_i = T_i^{\,i \in 1..n}\}$。如果一个代换 σ 的代换实例 σS 和 σT 相同,则称该代换合一了等式 S = T 。如果该代换能合一 C 中的所有等式,则称 σ 能合一(或满足)C。

22.3.2 定义: 约束类型关系 $\Gamma \vdash t:T \mid_{\mathcal{X}} C$ 的定义见参图 22.1 的规则。非正规地,$\Gamma \vdash t:T \mid_{\mathcal{X}} C$

① 还有许多其他建立此基本定义的方式。其中一种是用称为存在量词合一的一般机制,是由 Kirchner 和 Jouannaud (1990)提出的,以代替图 22.1 中的强制产生规则中的所有单独的新变量条件。另一个大的改进是由 Rémy(1992a, 1992b, long version, 1998, 第 5 章)提出,把类型声明本身当成合一体;从三元组 Γ, t, T)开始,其中三个分量都可能含类型变量,寻找代换 σ 使 $\sigma\Gamma \vdash \sigma(t):\sigma(T)$,即,该代换能使示意性的类型声明 $\Gamma \vdash t:T$ 合一化。

可读成“约束集C满足时，项 t 在 Γ 下的类型为 T”。在规则 T-App 中，以 FV(T)来表示 T 中提到的所有类型变量集合。

$$\frac{\mathtt{x}:\mathsf{T}\in\Gamma}{\Gamma\vdash\mathtt{x}:\mathsf{T}\ |_{\varnothing}\ \{\}}\quad\text{(CT-VAR)}$$

$$\frac{\Gamma,\mathtt{x}:\mathsf{T}_1\vdash\mathtt{t}_2:\mathsf{T}_2\ |_{\mathcal{X}}\ C}{\Gamma\vdash\lambda\mathtt{x}:\mathsf{T}_1.\mathtt{t}_2:\mathsf{T}_1\to\mathsf{T}_2\ |_{\mathcal{X}}\ C}\quad\text{(CT-ABS)}$$

$$\frac{\begin{array}{c}\Gamma\vdash\mathtt{t}_1:\mathsf{T}_1\ |_{\mathcal{X}_1}\ C_1\qquad\Gamma\vdash\mathtt{t}_2:\mathsf{T}_2\ |_{\mathcal{X}_2}\ C_2\\ \mathcal{X}_1\cap\mathcal{X}_2=\mathcal{X}_1\cap FV(\mathsf{T}_2)=\mathcal{X}_2\cap FV(\mathsf{T}_1)=\varnothing\\ \mathsf{X}\notin\mathcal{X}_1,\mathcal{X}_2,\mathsf{T}_1,\mathsf{T}_2,C_1,C_2,\Gamma,\mathtt{t}_1,\text{or }\mathtt{t}_2\\ C'=C_1\cup C_2\cup\{\mathsf{T}_1=\mathsf{T}_2\to\mathsf{X}\}\end{array}}{\Gamma\vdash\mathtt{t}_1\ \mathtt{t}_2:\mathsf{X}\ |_{\mathcal{X}_1\cup\mathcal{X}_2\cup\{\mathsf{X}\}}\ C'}\quad\text{(CT-APP)}$$

$$\Gamma\vdash 0:\mathsf{Nat}\ |_{\varnothing}\ \{\}\quad\text{(CT-ZERO)}$$

$$\frac{\Gamma\vdash\mathtt{t}_1:\mathsf{T}\ |_{\mathcal{X}}\ C\qquad C'=C\cup\{\mathsf{T}=\mathsf{Nat}\}}{\Gamma\vdash\mathtt{succ}\ \mathtt{t}_1:\mathsf{Nat}\ |_{\mathcal{X}}\ C'}\quad\text{(CT-SUCC)}$$

$$\frac{\Gamma\vdash\mathtt{t}_1:\mathsf{T}\ |_{\mathcal{X}}\ C\qquad C'=C\cup\{\mathsf{T}=\mathsf{Nat}\}}{\Gamma\vdash\mathtt{pred}\ \mathtt{t}_1:\mathsf{Nat}\ |_{\mathcal{X}}\ C'}\quad\text{(CT-PRED)}$$

$$\frac{\Gamma\vdash\mathtt{t}_1:\mathsf{T}\ |_{\mathcal{X}}\ C\qquad C'=C\cup\{\mathsf{T}=\mathsf{Nat}\}}{\Gamma\vdash\mathtt{iszero}\ \mathtt{t}_1:\mathsf{Bool}\ |_{\mathcal{X}}\ C'}\quad\text{(CT-ISZERO)}$$

$$\Gamma\vdash\mathtt{true}:\mathsf{Bool}\ |_{\varnothing}\ \{\}\quad\text{(CT-TRUE)}$$

$$\Gamma\vdash\mathtt{false}:\mathsf{Bool}\ |_{\varnothing}\ \{\}\quad\text{(CT-FALSE)}$$

$$\frac{\begin{array}{c}\Gamma\vdash\mathtt{t}_1:\mathsf{T}_1\ |_{\mathcal{X}_1}\ C_1\\ \Gamma\vdash\mathtt{t}_2:\mathsf{T}_2\ |_{\mathcal{X}_2}\ C_2\qquad\Gamma\vdash\mathtt{t}_3:\mathsf{T}_3\ |_{\mathcal{X}_3}\ C_3\\ \mathcal{X}_1,\mathcal{X}_2,\mathcal{X}_3\ \text{无交集}\\ C'=C_1\cup C_2\cup C_3\cup\{\mathsf{T}_1=\mathsf{Bool},\mathsf{T}_2=\mathsf{T}_3\}\end{array}}{\Gamma\vdash\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3:\mathsf{T}_2\ |_{\mathcal{X}_1\cup\mathcal{X}_2\cup\mathcal{X}_3}\ C'}\quad\text{(CT-IF)}$$

图 22.1 约束类型规则

下标$\mathcal{X}$是用来记录在每个子推导中出现的中间类型变量，以确保每个不同的子推导产生的新变量都不同。第一次阅读规则时，可以先不去管这些下标，以及与其有关的前提条件。再一次看规则时，可以研究一下这些注释和前提，做到两点：第一，在推导中，最终规则选择的类型变量必须与前面的子推导中选中的变量不同；第二，无论何时，若一个规则含两个或两个以上的子推导，这些子推导所选择的变量的集合之间不能有交集。还要留意这些条件不能阻止对给定的项做某些推导，它们只能阻止推导时在两个不同的地方使用同一个“新”变量，因为有无穷多的变量名可以使用，我们总能找到一个满足要求的新变量。

当我们自底向上读规则时，约束类型规则说明了一个直接的过程：对给定的 Γ 和 t，计算出 T 和 C(及$\mathcal{X}$)，满足 $\Gamma\vdash \mathtt{t}:\mathsf{T}|_{\mathcal{X}}C$。然而，不同于简单类型演算中的一般类型化算法，该过程总会成功，也就是说对每个 Γ 和 t 都存在 T 和C满足 $\Gamma\vdash \mathtt{t}:\mathsf{T}|_{\mathcal{X}}C$，而且该 T 和$C$是由 Γ 和 t 惟一决定的(严格地说，只当我们将其考虑为“对新名字的选择取模”时算法是确定的，这一点将在练习 22.3.9 中还会提到)。

为了减少下面讨论的符号量，有时可以省略$\mathcal{X}$，直接写成 $\Gamma\vdash \mathtt{t}:\mathsf{T}|C$。

22.3.3 练习[★ ↛]：用 S,$\mathcal{X}$,C来构造一个约束类型推导，使得结论为：

$$\vdash\lambda\mathtt{x}:\mathsf{X}.\ \lambda\mathtt{y}:\mathsf{Y}.\ \lambda\mathtt{z}:\mathsf{Z}.\ (\mathtt{x}\ \mathtt{z})\ (\mathtt{y}\ \mathtt{z}):\mathsf{S}\ |_{\mathcal{X}}\ C$$

约束类型化关系的思想是，对给定的项 t 和上下文 Γ，我们可以检查在 Γ 中 t 是否为可类型化的，方法是先找到能使 t 有类型的约束集C，以及一个可以反映出 t 的类型特征的结果类型 S；接着，为找到 t 的解，要找到满足C的代换 σ(就是能使C中的等式都相同)；对每个这样的代换 σ，类型 σS 就是 t 的可能类型。如果找不到满足C的代换，t 就没有使其可类型化的实例。

例如，对项 t = λx:X→Y.x 0 按照算法产生的约束集为{Nat→Z = X→Y}，相关的结果类型为(X→Y)→Z。代换 σ = [X ↦ Nat, Z ↦ Bool, Y ↦ Bool]使得等式 Nat→Z = X→Y 恒等，所以我们可推断 σ((X→Y)→Z)，即(Nat→Bool)→Bool 是 t 的可能类型。

以下的定义是对该思想的正式提出。

22.3.4　定义：设 $\Gamma \vdash t:S|C$。则 (Γ,t,S,C) 的解为序对 (σ,T)，其中 σ 满足 C 且 $\sigma S = T$。

下一节中将提出寻找使约束集 C 合一的代换算法问题。这里，首先要检查一下约束类型算法与原始声明的类型关系是否合理地对应。

假定有上下文 Γ 和项 t，为使 Γ 和 t 中的变量找到可能的实例而产生有效的类型，可以采取两个不同的特征方式：

1. [声明性的]根据定义(22.2.1)考虑用所有 (Γ,t) 的解的集合。
2. [算法性的]通过约束类型关系，找到满足 $\Gamma \vdash t:S|C$ 的 S 和 C，取 (Γ,t,S,C) 的解组成的集合。

可用两步来表明上面两个特征的等价性：首先，我们知道所有 (Γ,t,S,C) 的解都是 (Γ,t) 的解[参见定理(22.3.5)]；其次，通过对约束产生式的类型变量赋值，每个 (Γ,t) 的值都能扩充为 (Γ,t,S,C) 的值[参见定理(22.3.7)]。

22.3.5　定理[约束类型的可靠性]：设 $\Gamma \vdash t:S|C$，如果 (σ,T) 是 (Γ,t,S,C) 的解，则它也是 (Γ,t) 的解。

按照该结论，新变量集合 $\mathcal{X}$ 已不再重要，可以省略。

证明：通过对 $\Gamma \vdash t:S|C$ 的约束类型推导的归纳，结合最后使用的规则，几种情况分别讨论如下：

情况 CT-VAR:　$t = x$　$x:S \in \Gamma$　$C = \{\}$

已知 (σ,T) 是 (Γ,t,S,C) 的解，因为 C 为空，这样 $\sigma S = T$，但根据 T-Var，很快可得出 $\sigma\Gamma \vdash x:T$。

情况 CT-ABS:　$t = \lambda x{:}T_1.t_2$　$S = T_1 \to S_2$　$\Gamma, x{:}T_1 \vdash t_2 : S_2 \mid C$

已知 (σ,T) 是 (Γ,t,S,C) 的解，即 σ 使 C 合一且 $T = \sigma S = \sigma T_1 \to \sigma S_2$，因此 $(\sigma,\sigma S_2)$ 是 $((\Gamma,x{:}T_1),t_2,S_2,C)$ 的解。通过归纳假设可知 $(\sigma,\sigma S_2)$ 是 $((\Gamma,x{:}T_1),t_2)$ 的解，即 $\sigma\Gamma,x{:}\sigma T_1 \vdash \sigma t_2 : \sigma S_2$。通过 T-Abs，按要求得 $\sigma\Gamma \vdash \lambda x{:}\sigma T_1.\sigma t_2 : \sigma T_1 \to \sigma S_2 = \sigma(T_1 \to S_2) = T$。

情况 CT-APP:　$t = t_1\ t_2$　　$S = X$
$\Gamma \vdash t_1 : S_1 \mid C_1$　　$\Gamma \vdash t_2 : S_2 \mid C_2$
$C = C_1 \cup C_2 \cup \{S_1 = S_2 \to X\}$

根据定义，σ 使 C_1 和 C_2 合一且 $\sigma S_1 = \sigma(S_2 \to X)$。所以 $(\sigma,\sigma S_1)$ 和 $(\sigma,\sigma S_2)$ 是 (Γ,t_1,S_1,C_1) 和 (Γ,t_2,S_2,C_2) 的解，由此通过归纳假设可得 $\sigma\Gamma \vdash \sigma t_1 : \sigma S_1$ 和 $\sigma\Gamma \vdash \sigma t_2 : \sigma S_2$。但因为 $\sigma S_1 = \sigma S_2 \to \sigma X$，可得 $\sigma\Gamma \vdash \sigma t_1 : \sigma S_2 \to \sigma X$，根据 T-App 有 $\sigma\Gamma \vdash \sigma(t_1\ t_2) : \sigma X = T$ 其他情况类似。

对于一般的类型关系，对约束类型完备性的论证有点困难，因为必须对新名称小心处理。

22.3.6　定义：对所有在 $\mathcal{X}$ 中未定义的变量都用 $\sigma \backslash \mathcal{X}$ 作为代换，其他的仍以 σ 代换。

22.3.7　定理[约束类型的完备性]：设 $\Gamma \vdash t:S|_{\mathcal{X}}\ C$，如果 (σ,T) 是 (Γ,t) 的解且 $dom(\sigma) \cap \mathcal{X} = \varnothing$ 则有 (Γ,t,S,C) 的解 (σ',T) 满足 $\sigma' \backslash \mathcal{X} = \sigma$。

证明：根据约束类型推导归纳。

情况 CT-VAR: $t = x \quad x:S \in \Gamma$

从假设可知(σ,T)是(Γ,x)的解，从类型关系的逆转引理(9.3.1)可得到 $T=\sigma S$，但(σ,T)也是$(\Gamma,x,S,\{\})$的一个解。

情况 CT-ABS: $t = \lambda x:T_1.t_2 \quad \Gamma,x:T_1 \vdash t_2 : S_2 \mid_{\mathcal{X}} C \quad S = T_1 \to S_2$

根据假设可知(σ,T)是$(\Gamma,\lambda x:T_1.t_2)$的解，根据类型关系逆转引理可得出对某些 T_2 可产生 $\sigma\Gamma,x:\sigma T_1 \vdash \sigma t_2:T_2$ 和 $T=\sigma T_1\to T_2$。根据归纳假设，存在$((\Gamma,x:T_1),t_2,S_2,C)$的解$(\sigma',T_2)$且满足 $\sigma'\backslash\mathcal{X}$ 与 σ 一致。这时，$\mathcal{X}$不会包含任何 T_1 中的类型变量，所以 $\sigma'T_1=\sigma T_1$，且 $\sigma'(S) = \sigma'(T_1\to S_2)=\sigma T_1\to\sigma'S_2=\sigma T_1\to T_2=T$。这样我们认为$(\sigma',T)$是$(\Gamma,(\lambda x:T_1.t_2),T_1\to S_2,C)$的一个解。

情况 CT-APP: $t = t_1\ t_2 \quad \Gamma \vdash t_1 : S_1 \mid_{\mathcal{X}_1} C_1 \quad \Gamma \vdash t_2 : S_2 \mid_{\mathcal{X}_2} C_2$

$\mathcal{X}_1 \cap \mathcal{X}_2 = \varnothing$

$\mathcal{X}_1 \cap FV(S_2) = \varnothing$

$\mathcal{X}_2 \cap FV(S_1) = \varnothing$

X 未包含在 $\mathcal{X}_1, \mathcal{X}_2, S_1, S_2, C_1, C_2$ 中

$S = X \quad \mathcal{X} = \mathcal{X}_1 \cup \mathcal{X}_2 \cup \{X\} \quad C = C_1 \cup C_2 \cup \{S_1 = S_2 \to X\}$

从假设可知(σ,T)是$(\Gamma,t_1\ t_2)$的解，根据类型关系逆转引理可得 $\sigma\Gamma\vdash\sigma t_1:T_1\to T$ 和 $\sigma\Gamma\vdash\sigma t_2:T_1$。通过归纳假设可得$(\sigma_1,T_1\to T)$为$(\Gamma,t_1,S_1,C_1)$ 的解，(σ_2,T_1)为(Γ,t_2,S_2,C_2)，且 $\sigma_1\backslash\mathcal{X}_1=\sigma=\sigma_2\backslash\mathcal{X}_2$。我们必须表达出一个代换 σ'，使之满足：(1)$\sigma'\backslash\mathcal{X}$与 σ 一致；(2)$\sigma'X=T$；(3)σ'分别使 C_1,C_2 合一；(4)σ'使$\{S_1=S_2\to X\}$ 合一，即 $\sigma'S_1=\sigma'S_2\to\sigma'X$。这样可定义 σ'为如下形式：

$$\sigma' = \left[\begin{array}{lll} Y \mapsto U & \text{若 } Y \notin \mathcal{X} \text{ 且 } & (Y \mapsto U) \in \sigma, \\ Y_1 \mapsto U_1 & \text{若 } Y_1 \in \mathcal{X}_1 \text{ 且 } & (Y_1 \mapsto U_1) \in \sigma_1, \\ Y_2 \mapsto U_2 & \text{若 } Y_2 \in \mathcal{X}_2 \text{ 且 } & (Y_2 \mapsto U_2) \in \sigma_2, \\ X \mapsto T & & \end{array}\right]$$

条件(1)和条件(2)显然满足。因为$\mathcal{X}_1$ 和$\mathcal{X}_2$ 没有交集，所以条件(3)也满足。对新变量的附加条件可以保证 $FV(S_1)\cap(\mathcal{X}_2\cup\{X\})=\varnothing$，因此 $\sigma'S_1=\sigma_1S_1$，接着计算如：$\sigma'S_1=\sigma_1S_1=T_1\to T=\sigma_2S_2\to T=\sigma'S_2\to\sigma'X=\sigma'(S_2\to X)$。

故条件(4)也满足。其他情况类似。

22.3.8 推论: 设 $\Gamma\vdash t:S\mid C$。当且仅当(Γ,t,S,C)有解时，(Γ,t)有解。

证明: 可由定理(22.3.5)和定理(22.3.7)得出。

22.3.9 练习[推荐,★★★]: 在编译过程中，由于规则 CT-App 对新类型变量的取名无法确定，可采用别的方式来解决这个问题，如调用一个产生新的类型变量的函数，每次调用时产生的新类型变量都不同于以前产生的变量，且不同于上下文中出现的所有类型变量或正在检查的项。因为全局 gensym 操作对隐藏的全局变量有副作用，所以它们正常推理较困难。但我们能通过约束产生规则来整理出一系列的未使用过的变量名，以一个非常精确和数学上易处理的方式模仿 gensym 操作。

用 F 来表示一系列不同的类型变量名。不用 $\Gamma \vdash t:T|_{\mathcal{X}}\ C$ 来作为约束产生关系，改用 $\Gamma \vdash_F t:T|_{F'} C$，其中 Γ，F 和 t 为算法的输入，T，F' 和 C 为输出。当需要一个新的类型变量时，算法就会去掉 F 的前部分而将 F 的其余部分作为 F' 返回。
请写出关于此算法的规则。并用合理的方式证明它与一般的约束产生规则是等价的。

22.3.10 练习[推荐,★★]:在 ML 中运行练习(22.3.9)的算法，使用下面的数据类型：

```
type ty =
    TyBool
  | TyArr of ty * ty
  | TyId of string
  | TyNat
```

作为类型，将：

```
type constr = (ty * ty) list
```

作为约束集。还需要一个可以表示新变量名字的无穷系列的方式。有很多方法可以做到这一点，通过使用下面的递归数据类型可直接实现这一点：

```
type nextuvar = NextUVar of string * uvargenerator
and uvargenerator = unit → nextuvar

let uvargen =
  let rec f n () = NextUVar("?X_" ^ string_of_int n, f (n+1))
  in f 0
```

其中 uvargen 是一个函数，当用()调用时，会返回形为 NextUVar(x,f)的值，其中 x 是一个新类型名称，f 是相同形式的另一个函数。

22.3.11 练习[★★]:请写出如何扩充约束产生类型算法，以处理一般的递归函数定义(参见 11.11 节)。

22.4 合一

为计算约束集的解，我们根据 Hindley (1969)和 Milner(1978)的思想用合一(Robinson,1971 提出)来检验解集是否非空，如果非空，则可以找出一个“最好”的元素使得所有的解都能从该元素中直接产生出来。

22.4.1 定义:如果对某代换 σ 使得代换 σ 和代换 σ' 有 $\sigma' = \gamma \circ \sigma$，则称代换 γ 比代换 σ' 更具一般性，写为 $\sigma \sqsubseteq \sigma'$。

22.4.2 定义:对约束集 C 的主合一子(或称为最一般合一子)是代换 σ，它能满足 C 且对所有满足 C 的代换 σ' 都有 $\sigma \sqsubseteq \sigma'$。

22.4.3 练习[★]:为下列的约束集写出最一般合一子(如果存在的话)：

{X = Nat, Y = X→X}　　{Nat→Nat = X→Y}
{X→Y = Y→Z, Z = U→W}　　{Nat = Nat→Y}
{Y = Nat→Y}　　{} (约束集为空)

22.4.4　定义：类型合一算法在图 22.2 定义[①]。图 22.2 中第二行语句"let $\{S=T\}\cup C'=C$"应理解成"从集合 C 中选择一个约束 $S=T$，并用 C' 表示 C 中的其余部分"。

```
unify(C)  =  if C = ∅, then [ ]
             else let {S = T} ∪ C' = C in
               if S = T
                 then unify(C')
               else if S = X and X ∉ FV(T)
                 then unify([X ↦ T]C') ∘ [X ↦ T]
               else if T = X and X ∉ FV(S)
                 then unify([X ↦ S]C') ∘ [X ↦ S]
               else if S = S1→S2 and T = T1→T2
                 then unify(C' ∪ {S1 = T1, S2 = T2})
               else
                 fail
```

图 22.2　合一算法

第 5 行的附加条件 $X \notin FV(T)$ 和第 7 行的 $X \notin FV(S)$ 可看成是发生检查。作用是防止算法产生一个含循环代换(如 $X \mapsto X\to X$)的解，因为循环代换会使得我们谈到的有穷类型表达毫无意义(如果将语言扩充到无穷类型表达，即和第 20 章和第 21 章一样含递归类型，则该发生检查可以忽略掉)。

22.4.5　定理：合一算法总能终止，当输入一个不能合一的约束集时，算法才会失败，否则就会返回一个主合一子。形式化表达为：

1. 对所有的 C，无论是失败还是返回一个代换，*unify*(C)都会终止。
2. 如果 *unify*(C) $=\sigma$，则 σ 就是 C 的合一子。
3. 如果 δ 是 C 的合一子，那么 *unify*(C) $=\sigma$ 满足 $\sigma \sqsubseteq \delta$。

证明：对第(1)条，首先为约束集 C 定义一个称为度的序对(m,n)，其中 m 表示 C 中不同类型变量的数量，n 表示 C 中类型的总长。很明显 *unify* 算法的每条语句或者很快终止(第一种情况就成功或到最后一种失败)，或者用一个按字典顺序且小一些的约束集来递归调用 *unify*。

第(2)条是在计算 *unify*(C)的过程中对产生的递归调用数目的直接归纳。除了两个包含的变量外其他的情况都不重要，这要取决于下面的考虑：如果 σ 使$[X \mapsto T]D$ 合一，那么对任意约束集 D，$\sigma\circ[X \mapsto T]$都能使$\{X=T\}\cup D$ 合一。

第(3)条也是在计算 *unify*(C)的过程中对产生的递归调用数目的直接归纳。如果 C 是空集，则 *unify*(C)立即返回一个空代换[ ]；因为 $\delta=\delta\circ[\,]$，所以$[\,]\sqsubseteq\delta$ 满足要求。如果 C 非空，则 *unify*(C)会从 C 中选择某序对(S,T)，继续按照 S 和 T 的具体情况来处理：

情况：　$S=T$

因为 δ 是 C 的合一子，它也同样能合一 C'。通过归纳假设，有 *unify*(C) $=\sigma$ 满足 $\sigma\sqsubseteq\delta$。

情况：　$S=X$ 且 $X\notin FV(T)$

① 该算法中并不是说所得的合一类型表达是与其他种类的表达唱反调，相同的算法也能在任何(一阶)的表达之间得出相等的约束集。

因为 δ 能使 S 和 T 合一,所以有 δ(X) = δ(T)。所以对任意类型 U,都有 δ(U) = δ([X ↦ T] U);尤其是因为 δ 能合一 C',也必能使 [X ↦ T] C'合一。通过归纳假设可知,*unify*([X ↦ T] C') = σ′,其中存在 γ 满足 δ = γ∘σ′。因为 *unify*(C) = σ′∘[X ↦ Y],表明 δ = γ∘(σ′∘[X ↦ T]) 能完成论证。所以考虑任意类型的变量 Y,如果 Y≠X,显然(γ∘(σ′∘[X ↦ Y]))Y = (γ∘σ′)Y = δY。另一方面,从上面可看出(γ∘(σ′∘[X ↦ T]))X = (γ∘σ′)T = δX。通过以上所述,可知对所有的变量 Y 都有 δY = (γ∘(σ′∘[X ↦ T]))Y,即 δ = (γ∘(σ′∘[X ↦ T]))。

情况: T = X 且 X ∉ *FV*(S)

情况类似。

情况: $S = S_1 \to S_2$ 且 $T = T_1 \to T_2$

可直接得证。

另外要注意当且仅当 δ 是 $C' \cup \{S_1 = T_1, S_2 = T_2\}$ 的合一算子时,δ 也是 $\{S_1 \to S_2 = T_1 \to T_2\} \cup C'$ 的合一算子。

如果上面的情况都不适用于 S 和 T,则 *unify*(C)失败。但这种情况只会有两种原因:或者 S 是 Nat 型,T 是箭头类型(或两者反过来); 或者 S = X,X∈T(或两者反过来)。第一种情况明显不符合 C 是可合一的这个假设,所以只能考虑第二种情况。假设第二种情况发生,则 δS = δT; 如果 X 包含在 T 中,则 δT 总会严格地比 δS 大。于是,如果 *unify*(C)失败,则 C 是不可合一的,但这又与我们开始的假设 δ 是 C 的合一算子矛盾,所以,这种情况也不可能发生。

22.4.6 练习[推荐,★★★]:实现合一化算法。

22.5 主类型

上面我们曾说过如果有某种方法使项中的类型变量实例化,从而使项本身成为可类型化的,则可以找到一个最一般化或主要的方法来说明,我们要就此提出形式化的理论。

22.5.1 定义:(Γ,t,S,C)的主解是解(σ,T),若任何时候有(σ′,T′)也是(Γ,t,S,C)的解,则有 σ ⊑ σ′。当(σ,T)是一个主解时,称 T 是 Γ 条件下的 t 的主类型[①]。

22.5.2 练习[★ ↛]:为 λx:X.λy:Y.λz:Z.(x z)(y z)找出主类型。

22.5.3 定理[主类型]:如果(Γ,t,S,C)有解,则它必有一个主解。图 22.2 中的合一算法能用来判断(Γ,t,S,C)是否有解,如果有,则可以将主解计算出来。

证明:可根据(Γ,t,S,C)解的定义及合一的性质来证明。

22.5.4 推论:(Γ,t)是否有解是可判断的

证明:根据推论(22.3.8)和定理(22.5.3)可得。

22.5.5 练习[推荐,★★★ ↛]:结合练习 22.3.10 和练习 22.4.6 的约束产生算法和合一算

① 请不要将主类型与主类型化混淆了。参见 22.8 节。

法来建立可计算主类型的类型检查器,以 reconbase 检查器作为起点。类型检查器的典型的作用过程类似如下:

```
λx:X. x;
▸ <fun> : X → X

λz:ZZ. λy:YY. z (y true);
▸ <fun> : (?X0→?X1) → (Bool→?X0) → ?X1

λw:W. if true then false else w false;
▸ <fun> : (Bool→Bool) → Bool
```

类型变量名如 $?X_0$ 是自动生成的。

22.5.6 练习[推荐,★★★]:如果要处理记录型而想扩充上面的定义(22.3.2 等)时会出现什么困难? 该怎样将它们提出?

主类型思想可以用于建立一个类型重构算法,它比这里所研究的算法运行更快。不是先产生所有的约束集然后再求解,而是将产生集和求解交叉进行,这样类型重构算法在每一步都能真正地返回一个主类型。因为类型总是主要的,所以算法不需要对子项再进行分析——这样对每一步只要花最少的力气就能达到可类型化的目的。这种算法的一个最大的进步就是它能更精确地指出用户程序的错误之处。

22.5.7 练习[★★★ ↛]:修改练习 22.5.5 的解,让它能更快地实现合一化并返回主类型。

22.6 隐含的类型注释

支持类型重构的语言还允许编程人员完全忽略 lambda 抽象类型注释。为做到这一点(如在 22.2 节所述)只要用解析器将新产生的类型变量来填充省略的注释即可。一个比较好的选择就是将未注释的抽象加入到项语法中,以及将一个对应的规则加入到约束类型关系中。

$$\frac{X \notin \mathcal{X} \qquad \Gamma, x{:}X \vdash t_1 : T \mid_{\mathcal{X}} C}{\Gamma \vdash \lambda x.t_1 : X \to T \mid_{\mathcal{X} \cup \{X\}} C} \qquad \text{(CT-AbsInf)}$$

对这些未注释的抽象说明不把它们当成语法的修饰品会更直接些。这是以简短但有用地使表达更有利的方式:如果复制几份未注释的抽象,CT-AbsInf 规则允许为每个副本选择一个不同的变量作为参数类型。相比之下,如果将原始抽象看成是用不可见的类型变量来注释,那么生成的副本将会产生多个含相同参数类型的表达。这点区别对下一节要讨论 let 多态十分重要。

22.7 let 多态

"多态"(polymorphism)一词是语言机制范围内的一个概念,它指的是程序的单独一段能在不同的上下文中使用不同的类型(23.2 节将更详细地讨论几种多态形式)。上面介绍的类型重构算法可以推广成一个简单的多态形式,称为 let 多态(也可称为 ML 型或 DamasMilner 多态)。该特征最早出现在 ML(Milner, 1978)的术语中,已经被许多成功的语言设计所采纳,它形

成常用结构(列表、数组、树型、哈希列表、流和用户接口的小窗口,等等)的基因库的基础部分。

let 多态的功能可从下面的例子谈起。假设要定义并使用一个简单函数 double,要求使用第一个参数两次后再使用第二个:

```
let double = λf:Nat→Nat. λa:Nat. f(f(a)) in
double (λx:Nat. succ (succ x)) 2;
```

因为想将 double 应用到类型为 Nat →Nat 的函数,可选择注释类型(Nat →Nat) →(Nat →Nat)。我们可变换 double 定义使一个布尔型函数加倍。

```
let double = λf:Bool→Bool. λa:Bool. f(f(a)) in
double (λx:Bool. x) false;
```

我们不能做的是在同一个 double 函数中既用布尔型又用数字型,如果需要这两者在同一个程序中,必须定义两个形式相同但类型注释不同的 double 函数。

```
let doubleNat = λf:Nat→Nat. λa:Nat. f(f(a)) in
let doubleBool = λf:Bool→Bool. λa:Bool. f(f(a)) in
let a = doubleNat (λx:Nat. succ (succ x)) 1 in
let b = doubleBool (λx:Bool. x) false in ···
```

即使在 double 中用类型变量来注释抽象也没有作用:

```
let double = λf:X→X. λa:X. f(f(a)) in ···
```

例如,如果写:

```
let double = λf:X→X. λa:X. f(f(a)) in
let a = double (λx:Nat. succ (succ x)) 1 in
let b = double (λx:Bool. x) false in ···
```

则在 a 定义中使用 double 会产生一个约束类型 X→X = Nat →Nat,而在 b 定义中使用 double 会产生约束类型 X→X = Bool→Bool,这样会使 X 发生矛盾而导致整个程序不可类型化。

这里到底出了什么错? 在例子中 X 承担了两个不同的角色。其一,它受到了约束,即计算 a 的 double 的第一个参数必须为一个定义域和值域类型与 double 的其他参数的类型(Nat)相同的函数。其二,它同样受到了 b 中的 double 类似的约束。然而,两种情况下都使用相同的变量 X,我们不得不终止得到的错误约束,即 double 的两次使用得到的第二参数必须有相同的类型。

这里该做的是打破最后一个联系,也就是说,对每个 double 的使用都用不同的变量 X。幸好这一点很容易做到。第一步改变 let 的通常类型规则,不是计算右端的 t_1 类型并在计算 t_2 的类型时用得出的 t_1 类型来作为囿变量 x 的类型:

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \texttt{let } x{=}t_1 \texttt{ in } t_2 : T_2} \qquad \text{(T-Let)}$$

而是用 t_1 代换 x,然后对该扩充的表达式进行类型检查:

$$\frac{\Gamma \vdash [x \mapsto t_1]t_2 : T_2}{\Gamma \vdash \texttt{let } x{=}t_1 \texttt{ in } t_2 : T_2} \qquad \text{(T-LetPoly)}$$

用同样的方法为 let 写一个约束类型规则:

$$\frac{\Gamma \vdash [x \mapsto t_1]t_2 : T_2 \;\;|_{\mathcal{X}}\; C}{\Gamma \vdash \texttt{let } x{=}t_1 \texttt{ in } t_2 : T_2 \;\;|_{\mathcal{X}}\; C} \qquad \text{(CT-LetPoly)}$$

总之，我们所做的就是修改了 let 类型规则使它们能实现一步求值：

$$\texttt{let } x\texttt{=}v_1 \texttt{ in } t_2 \longrightarrow [x \mapsto v_1]t_2 \qquad \text{(E-LetV)}$$

第二步是用 22.6 节的隐含注释过的 lambda 抽象来重新定义 double。

```
let double = λf. λa. f(f(a)) in
let a = double (λx:Nat. succ (succ x)) 1 in
let b = double (λx:Bool. x) false in ...
```

将 let 的约束类型规则(CT-LetPoly)与隐含注释的 lambda 抽象(CT-AbsInf)结合起来的确满足了我们的要求：CT-LetPoly 产生了 double 定义的两个副本，CT-AbsInf 给每个抽象分配了不同的类型变量。其余的部分仍按照通常的约束类型求解方法来做。

然而，在实际使用该方法之前仍需说明它存在的一些缺点。一个明显的缺点就是，如果我们不是碰巧在 let 式子中使用 let 囿变量，那么该定义将永远不会受到类型检查。例如，有程序：

```
let x = <utter garbage> in 5
```

将能通过类型检查器。可以给下面类型规则增加一个前提：

$$\frac{\Gamma \vdash [x \mapsto t_1]t_2 : T_2 \qquad \Gamma \vdash t_1 : T_1}{\Gamma \vdash \texttt{let } x\texttt{=}t_1 \texttt{ in } t_2 : T_2} \qquad \text{(T-LetPoly)}$$

并给 CT-LetPoly 增加一个对应的前提(保证 t_1 是良类型的)来解决这个问题。

还有一个有关的问题是，如果 let 式子出现很多次 let 囿变量，那么 let 定义的整个右端在每次发生时将受到检查，无论它是否包含隐含注释的 lambda 抽象。因为右端的本身会包含 let 囿，所以该类型规则会造成类型检查器所需的工作量是原项的指数倍。

为了避免重复类型检查，含 let 多态的语言在实际运行时使用了更好(尽管形式上是等价的)的类型规则的重新公式化方式。总而言之，在上下文 Γ 中对项 let $x = t_1$ in t_2 进行类型检查应按照以下步骤进行：

1. 使用约束类型规则为右端的 t_1 计算相关的类型 S_1 和集合 C_1。
2. 用合一规则为约束集 C_1 找到最一般化解 σ，并将 σ 用于 S_1(和 Γ)来获得 t_1 的主类型 T_1。
3. 对 T_1 中的其余变量进行一般化推广。如果 $X_1 \cdots X_n$ 是剩余的变量，写成 $\forall X_1 \cdots X_n . T_1$ 为 t_1 的主类型方案。

   警告：要注意不要将 Γ 中提到的 T_1 也进行一般化推广，因为这些对应着 t_1 与其语境之间实际的约束。例如，在：

   ```
   λf:X→X. λx:X. let g=f in g(x);
   ```

   中，我们不能对 g 中的类型 X→X 的变量 X 进行一般化，否则将会得到如下错误的程序：

   ```
   (λf:X→X. λx:X. let g=f in g(0))
     (λx:Bool. if x then true else false)
     true;
   ```

4. 我们对上下文进行扩充，为囿变量 x 记录类型为 $\forall X_1 \cdots X_n . T_1$，并开始对 t_2 进行类型检查。总之，现在上下文会给每个自由变量一个类型方案，而不是一个类型。

5. 每次在 t_2 中遇到变量 x 发生时，我们可以查找它的类型方案 $\forall X_1 \cdots X_n . T_1$。现在产生新类型变量序列 $Y_1 \cdots Y_n$，并用它们来实例化类型方案，即产生 $[X_1 \mapsto Y_1, \cdots, X_n \mapsto Y_n]T_1$，以此作为 x 的类型①。

该算法比起在类型检查之前简单的采取代换 let 表达式的方法要高效得多。的确，数十年的实践表明该算法对输入程序的长度上呈现"基本线性关系"。所以当 Kfoury, Tiuryn 和 Urzyczyn(1990)三人及 Mairson(1990)都提出了该算法的最坏复杂度仍带指数时，引起了很大的震惊。他们所举的例子包括了在其他 let 的右端层嵌套 let 语句，而不是在 let 语句中，在 let 语句中很容易造成建立的表达式中，类型按指数增长而远远超过表达式本身。例如，下面的 OCaml 程序，按照 Mairson(1990)所述，该程序是良类型的，但类型检查过程需要大量的时间：

```
let f0 = fun x → (x,x) in
  let f1 = fun y → f0(f0 y) in
    let f2 = fun y → f1(f1 y) in
      let f3 = fun y → f2(f2 y) in
        let f4 = fun y → f3(f3 y) in
          let f5 = fun y → f4(f4 y) in
            f5 (fun z → z)
```

如果想知道为什么，可试着一次输入一个 $f_0, f_1, \cdots$，到 OCaml 最高层中。还可见 Kfoury, Tiuryn 和 Urzyczyn(1994)对此的深入讨论。

最后值得一提的是，用 let 多态来设计成熟的程序设计语言，还要注意多态与某些副作用，如易变存储单元之间的相互影响。一个简单的例子可说明其中的危害：

```
let r = ref (λx. x) in
(r:=(λx:Nat. succ x); (!r)true);
```

用上面所述的算法，计算 Ref(X→X)为 let 右端的主类型，因为 X 不会出现在其他的地方，所以该类型可被一般化为 ∀X.Ref(X→X)，将 r 加入到上下文时可以将该类型方案分配给它。对第二行的分派进行类型检查时，我们又将该类型实例化为 Ref(Nat→Nat)，对第三行可实例化为 Ref(Bool→Bool)。但这不太合理，因为当项在求值时将 succ 应用于 true 时就会结束。

这里的问题在于类型规则已经与求值规则不一致了。本节介绍的类型规则告诉我们，当看到 let 表达时，应当立即将右端代换到表达体中。但求值规则表明应当在右端简化为一个值之后才能进行代换。类型规则对 ref 构件有两种使用方式，并在不同的假设条件下分析它们，但在运行时只有一个 ref 是真正被分配使用的。

出现这种搭配不当时，有两种方法来纠正：或者调整求值规则；或者调整类型规则。前一种方法可将求值规则中的 let 变成②：

$$\texttt{let } x\texttt{=}t_1 \texttt{ in } t_2 \longrightarrow [x \mapsto t_1]t_2 \qquad \text{(E-Let)}$$

用这种方法，在对上面的例子进行求值的第一步是用它的定义来代替 r，产生：

---

① 一旦提到一般化和实例化时，一个明确类型变量注释的 lambda 抽象与一个需要约束产生算法来产生变量的未注释抽象之间的区别就会变得不再重要。任一种方法都会使 let 的右端被分配一个带变量的类型，该变量在加入上下文之前都是一般化的，且在每次实例化时都要被新变量代替。

② 严格地说，应该用一个存储器来注释该规则，与在第 13 章的做法一样，因为我们正在讨论带引用的语言：

$$\texttt{let } x\texttt{=}t_1 \texttt{ in } t_2 \mid \mu \longrightarrow [x \mapsto t_1]t_2 \mid \mu \qquad \text{(E-Let)}$$

```
(ref (λx. x)) := (λx:Nat. succ x) in
(!(ref (λx. x))) true;
```

该式是十分安全的！第一行产生了一个初始含相同函数的引用单元，将(λx:Nat.succ x)存放在里面；第二行产生了另一个包含相同函数的引用单元，提取出里面的内容并作用到 true。然而，这种计算也表明改变求值规则使之适合类型规则的做法，会产生不寻常的语义，即不再是按照标准的值调用求值顺序[带非 CBV 求值方法的命令式语言并不是前所未闻的(Augustsson, 1984)，但它们从未流行过，就因为在运行时对副作用的顺序的理解和控制存在困难]。

相比之下，改变类型规则，使之配合求值规则的方法更好些。因为十分容易，只要强加一个限制(通常称为值限制)就可使 let 囿可多态处理，也就是说，只要它的右端是一个语法值，它的自由类型变量可以一般化。比如，在有错误的例子中，当我们将 r 加入到上下文时，赋给它的类型是 X→X，而不是 ∀X.X→X。第二行所加的约束将会使 X 强行转换为 Nat，因为 Nat 不可能和 Bool 合一，这样就会造成第三行的类型检查失败。

用值限制的方法可以使类型更安全，只是在表述上会繁琐一些：我们再不能写这样一种程序——程序中 let 表达式的右端既能实现有趣的计算，又能被赋予一个多态类型。令人惊讶的是，这种限制不会在实际使用时产生任何不同之处。Wright(1995)通过分析大量用 ML 术语(ML 标准定义，由 Milner，Tofte 和 Harper 1990 年提出)写的文献代码(提供了基于弱类型变量的更灵活的 let 类型规则)而得出了这个结论，并发现无论如何几乎只有一小部分的右端是语法值。该结论或多或少结束了争论，现在所有的 ML 型的 let 多态都采纳值限制。

22.7.1 练习[★★★ ↛]：请实现本节的算法。

22.8 注释

关于 lambda 演算中的主类型概念至少要追溯到 20 世纪 50 年代的 Curry 的论著(Curry 和 Feys，1958)。基于 Curry 思想的计算主类型算法是由 Hindley(1969)提出的；类似的算法是由 Morris(1968)和 Milner(1978)独立提出的。在命题逻辑世界中，该思想提出得更早，大概是 20 世纪 20 年代的 Tarski 和 20 世纪 50 年代的 Meredith 兄弟(Lemmon，Meredith，Meredith，Prior 和 Thomas，1957)；算法第一次是由 David Meredith 于 1957 年在计算机上运行的。另外历史上对主类型的评论还有 Hindley(1997)。

合一(Robinson，1971)是计算机科学的许多领域的基础。可找到关于它的大量介绍，例如，在 Baader 和 Nipkow(1998)，Baader 和 Siekmann(1994)，Lassez 和 Plotkin(1991)都提到过。

ML 型的 let 多态首先是由 Milner(1978)提出的，那时大量的类型重构算法已经提出了，著名的为 Damas 和 Milner(1982；还有 Lee 和 Yi，1998)提出的经典的 W(Damas 和 Milner)算法。W 算法与本章提出的算法，其主要区别是前者提出的是"纯类型重构"——将主类型指派给完全无类型的 lambda 项，而本章是将类型检查和类型重构混在一起，允许项包含明确的类型注释，注释中可以(但不是一定)包含变量。这一点使得我们在技术表述上存在多一些的困难[尤其是定理(22.3.7)完备性的证明，因为在这里必须要谨慎地保持程序人员的类型变量与约束产生规则中提到的变量区别开来]，但它与其他章节的风格紧密配合。

一篇由 Cardelli(1987)写的经典论文列出了许多实现上的观点。Appel(1998)，Aho 等，(1986)和 Reade(1989)等人对其他类型重构算法进行了一些探讨。一篇十分精彩的，关于核心

系统(称为 mini-ML)的文章(由 Clement, Despeyroux, Despeyroux 和 Kahn, 1986 年撰写)形成了理论讨论的基础。Tiuryn(1990)对类型重构一系列问题进行了研究。

主类型不应该与相似概念主类型化混淆在一起。其区别是,当我们计算主类型时,上下文 Γ 和项 t 均看成是算法的输入,而主类型 T 是输出。计算主类型化方式的算法是只将 t 作为输入,将 Γ 和 T 视为输出,也就是说,它计算的是 t 中关于自由变量类型的最小假设。主类型化对支持独自汇编和"灵敏重汇编"、实现递增类型推论和指出类型错误方面都有作用。但遗憾的是,许多语言,尤其是 ML 语言,只有主类型而没有主类型化,参见 Jim(1996)。

ML 多态,由于其功能强大但又简易而令人瞩目,在程序设计领域占据一块"风水宝地",如果将其与其他复杂的类型特征结合,将变得更加的精妙。在这块领域中最大的成功就是为记录型进行类型重构的一个精彩陈述,是由 Wand(1987)提出的,以后 Wand(1988, 1989b)又对其进行了进一步的探讨,还有 Remy(1989, 1990; 1992a, 1992b, 1998)和其他许多学者都深入研究过。其中的思想是引入了一个新的变量,称为行变量,它不是类型的变化,而完全是字段标签和相关类型的"行"。相等合一的简单形式可用来解决含行变量的约束集问题,参见练习 22.5.6。Garrigue(1994)和其他一些人研究过动态类型的相关方法。这些技术已经拓展到一般性的概念,如类型类(Kaes, 1988; Wadler 和 Blott, 1989)、约束类型(Odersky, Sulzmann 和 Wehr, 1999),以及合格类型(Jones, 1994b, a),这些概念形成了 Haskell 的类型类系统的基础(Hall 等, 1996; Hudak 等, 1992; Thompson, 1999);类似的思想, Mercury(Somogyi, Henderson 和 Conway, 1996)和 Clean(Plasmeijer, 1998)都提出过。

Wells(1994)提出对功能更大的,不可预言多态形式的类型重构是不可决定的,关于不可预言多态将在第 23 章讨论。的确,该系统的一些部分类型重构的形式也证实是不可决定的。23.6 节和 23.8 节将对此结论提供更多的说明,还要就如何将 ML 型类型重构与多态更强的形式,如 2 秩多态结合起来的方法问题进行仔细说明。

对于子类型化与 ML 型类型重构的组合,一些有效的初始成果已经由众多学者得出(Aiken 和 Wimmers, 1993; Eifrig, Smith 和 Trifonov, 1995; Jagannathan 和 Wright, 1995; Trifonov 和 Smith, 1996; Odersky, Sulzmann 和 Wehr, 1999; Flanagan 和 Felleisen, 1997; Pottier, 1997),但实用的检查器还期待着推广使用。

扩充 ML 型的类型重构,使之能处理递归类型(参见第 20 章),并不是件十分困难的事(Huet, 1975, 1976)。对本章提出的算法惟一最困难的是对合一的定义,因为这里我们忽略了发生检查(这样做,通常是为保证合一算法返回的代换是无循环的)。这样做了以后,为确保能在有限时间内结束,还要修改合一算法中使用的表示方式,以使它能继续保持共享,也就是说,要使用破坏(隐含循环的)指针结构的操作。

另一方面,将类型重构与递归定义的项混淆了,会带来一个棘手的问题,称为多态递归。ML 中一个简单(且不存在问题)用来定义递归函数的类型规则能说明一个递归函数只能在自己的定义体内单态运行(即所有的递归调用必须是相同的类型参数和结果),而在程序的其余部分的发生可以是多态的(用不同类型的参数和结果)。Mycroft(1984)和 Meertens(1983)为递归定义提出一个多态类型规则,允许递归调用一个自身定义体中递归函数时可用不同的类型实例化。这种扩充,常常称为 Milner-Mycroft 演算,却被 Henglein(1993)以及 Kfoury, Tiuryn 和 Urzyczyn(1993a)等人提出过(存在着不可决定的重构问题);这两方面提出的证明都是依据于 Kfoury, Tiuryn 和 Urzyczyn(1993b)提出的(无限制)半合一问题的不可决定性。

第23章 全称类型①

在前一章中，我们学习了简单 ML 中的 let 多态。从本章开始，将要考虑在一个功能更强大的演算（称为系统 F）中设置多态的更一般形式。

23.1 动机

如在 22.7 节中所述，我们可在简单类型的 lambda 演算中写一个无限多的"加倍"（doubling）函数：

```
doubleNat = λf:Nat→Nat. λx:Nat. f (f x);
doubleRcd = λf:{l:Bool}→{l:Bool}. λx:{l:Bool}. f (f x);
doubleFun = λf:(Nat→Nat)→(Nat→Nat). λx:Nat→Nat. f (f x);
```

每个函数都用不同的参数类型，但所有的都实现相同的行为（的确，若不考虑类型注释，它们共用相同的程序语句）。如果我们想在相同的程序中运行不同参数类型说明的 doubling 函数，就应该为每个 T 写出各自的 doubleT 定义。这种剪切－粘贴的编程方式违反了软件工程的一个基本的规则：

规则提要：程序中的每个有意义的功能块都应该在源代码的同一个位置运行。对不同代码段实现类似功能的，可以将它们放在相同代码中仅提取出不同的部分。

这里提到的不同部分是指类型！我们所需要的是能够从项中提取出类型，并能在后面用具体类型注释来实例化这些抽象的项的工具。

23.2 各种多态

如果类型系统允许一段程序能使用多种类型，则称这种系统为多态系统（poly 意思为多，morph 为形式）。多态形的几种方式可在现代语言中找到（这种分类是由 Strachey，1967 和 Cardelli 及 Wegner，1985 提出的）。

参数化多态：本章的主题，允许一段代码能带一般化类型，即先用变量来代替实际类型，在需要时再用特殊的类型来实例化。参数定义都用合一式：它们的所有实例都实现相同的操作。

参数化多态功能最强大的形式是本章提出的不可预言或一级多态。实际使用中最常用的形式是 ML 型或称为 let 多态形，它可将多态形限制到最高层 let 囿，不允许将多态值作为参数的函数，但可以获得自动类型重构一个方便且自然的形式（参见第 22 章）。目前一级参数化多态形也开始在编程语言中流行起来，并成为了某些语言系统，如 ML（参见 Harper 和 Stone，2000）的强大模块的技术基础。

---

① 本章中学习的系统大部分是纯系统 F（参见图 23.1）。23.4 节中例子对前面学习过的特点做了扩充。相关的 OCaml 实现是 fullpoly（例子中涉及序对、列表都需要 fullomega 检查器）。

相比之下,权宜多态允许一个多态值在遇到不同的类型时能产生不同的操作。权宜多态最常见的例子是重载,即使用单一的函数形式去实现许多操作,编译器(或运行系统,取决于重载归结是静态的还是动态的)根据参数的类型,为函数的每个应用选择合适的实现方式。

函数重载概念的出现形成了语言的多方式处理的基础,比如语言 CLOS (Bobrow 等,1988; Kiczales 等, 1991) 和 Cecil (Chambers, 1992; Chambers 和 Leavens, 1994)。这一原理已经在 Castagna, Ghelli 和 Longo(1995; Castagna, 1997)提出的 λ-& 演算中形式化了。

权宜多态的一个更强形式称为加强多态(Harper 和 Morrisett, 1995; Crary, Weirich 和 Morrisett, 1998),允许在运行时对类型进行受限计算。对于许多多态语言的高级实现而言,加强多态是一种有效的技术,包括无标记垃圾收集、提取函数参数、多态排列和空间有效的"平整"数据结构。

然而,权宜多态可通过 typecase 原语构造一个强形式,它支持运行时根据类型信息进行任意模式匹配(Abadi, Cardelli, Pierce 和 Rémy, 1995; Abadi, Cardelli, Pierce 和 Plotkin, 1991b; Henglein, 1994; Leroy 和 Mauny, 1991; Thatte, 1990)。语言特性如 Java 中的 instanceof 测试就可看成是 typecase 的受限形式。

第 15 章的子类型多态通过假定的规则给一个项更多的类型,这样我们可以有选择地去"忘记"项的具体操作。

以上这些分类并不是彼此孤立的,不同形式的多态可以在同一个语言中出现。如标准 ML 既提供参数化多态又提供简单的嵌入式算术操作重载,但不包括子类型,而 Java 语言包含子类型、重载和简单的权宜多态(如 instanceof),但不包含参数化多态(至少到写稿为止还没有)。有许多想把参数化多态加到 Java 语言中的提议,其中最著名的是 GJ(Bracha, Odersky, Stoutamire 和 Wadler, 1998)。

光说"多态"是不负责的行为,因为这样会造成编程语言领域之间很大的混乱。在功能性编程人员(即那些使用或设计如 ML, Haskell 等语言的人员)中,多态指的通常都是参数化多态。而另一方面,对面向对象的编程人员来说,它指的是子类型多态,将术语 genericity(或 generic)用于参数化多态。

23.3 系统 F

本章将要学习的系统通常称为系统 F,它首次由 Jean-YvesGirard(1972)在逻辑证明理论的上下文中发现。过后不久,一个基本功能相同的类型系统由一位叫做 John Reynolds(1974)的计算机科学家独立开发出来,他还把该系统称为多态 lambda 演算。该系统已经作为多态基础研究的一个工具和作为许多编程语言设计基础而广泛采纳。有时该系统还被称为二阶 lambda 演算,因为它通过 Curry-Howard 对应,与二阶直觉逻辑相对应,这种逻辑不仅对个体(项)进行量化,还对谓词(类型)进行量化。

系统 F 的定义是对简单类型的 lambda 演算——$\lambda_{\to}$ 的推广。在 $\lambda_{\to}$ 中,lambda 抽象是从项中提取项,应用是为抽象部分赋值。因为我们需要先从项中抽象出类型,稍后再将类型添加进去,所以引入了一个新的抽象形式,写为 $\lambda X.t$,它的参数是一个类型和一个新的应用形式 $t[T]$,在 $t[T]$ 中的参数是一个类型表达式。我们把新的抽象称为类型抽象,新的应用构造为类型应用或实例化。

在求值过程中，当一个类型抽象遇到一个类型应用时，它们形成了一个约式(redex)，如在$\lambda_{\to}$中一样。添加一个归约规则：

$$(\lambda X.t_{12})\ [T_2] \longrightarrow [X \mapsto T_2]t_{12} \qquad \text{(E-TappTabs)}$$

它类似于一般的抽象和应用归约规则：

$$(\lambda x{:}T_{11}.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \qquad \text{(E-AppAbs)}$$

例如，当多态恒等函数：

```
id = λX. λx:X. x;
```

应用于类型 Nat 时，记成 id[Nat]时，结果为$[X \mapsto Nat](\lambda x{:}X.x)$，也就是 $\lambda x{:}Nat.x$，为自然数上的恒等函数。

最后，还要说明一个多态抽象的类型。用类型，如 Nat→Nat 来分类一般函数 $\lambda x{:}Nat.x$；现在还需要一个定义为某类型的"箭头类型"的不同形式，用来分类多态函数，如 id。注意，对其使用的每个参数 T，id 会产生一个类型为 T→T 的函数；也就是说，id 结果的类型是依赖于作为参数传递的实际类型。为说明这种依赖性，将 id 的类型写成$\forall X.X\to X$。多态抽象和应用的类型化规则类似于项抽象和应用规则。

$$\frac{\Gamma, X \vdash t_2 : T_2}{\Gamma \vdash \lambda X.t_2 : \forall X.T_2} \qquad \text{(T-TAbs)}$$

$$\frac{\Gamma \vdash t_1 : \forall X.T_{12}}{\Gamma \vdash t_1\ [T_2] : [X \mapsto T_2]T_{12}} \qquad \text{(T-TApp)}$$

注意，在对 t 的子推导上下文中加入了类型变量 X。我们仍要按照定义(5.3.4)的约定，即(项或类型)变量的名称的选择应该与 Γ 中已出现过的不同，为满足这一条件，lambda 囿的类型变量也可相应重命名(在关于系统 F 的某些陈述中，这个新条件是作为 T-TAbs 规则的外在附加条件，而不是把它加入到构造上下文的规则当中，这一点与我们这里的做法不同)。此时，上下文中类型变量的惟一作用就是记录演算的中间过程，确保相同的类型变量不会两次加入到同一上下文中。在后面的章节中，我们将用多种信息来注释类型变量，如囿(参见第 26 章)和分类(参见第 29 章)。

图 23.1 是多态 λ 演算的完整定义，突出了与$\lambda_{\to}$的区别。通常，该图只定义了纯演算，忽略了其他类型结构，如记录型、基类型(Nat 和 Bool)，以及项语言扩展(如 let 和 fix)。但这些特殊的结构可以直接加到纯系统中，我们在接下来的例子中会随时使用它们。

23.4 实例

现在我们研究几个带多态的程序实例。为激发大家的兴趣，这里先举几个小的但越来越难解决的例子，以此来说明系统 F 表达上的某些优势。然后要重温一下带列表、树等"普通"多态程序的基本思想。最后两小节将介绍简单的代数数据类型，如布尔型、数字型和列表型的 Church 编码的类型形式，其中 Church 编码在第 5 章的无类型 lambda 演算中出现过。尽管这些编码在实际中起不到大作用，但如果将它们作为原语植入高级编程语言中，会使编译好的代码更加容易，因为它们对理解系统 F 的复杂性和功能有很大的帮助。在第 24 章中，我们还能看到多态在模块式程序设计和抽象数据类型等领域的应用。

$\rightarrow \forall$ 基于 $\lambda_{\rightarrow}$ *(9.1)*

语法

t ::=		项：
	x	变量
	$\lambda x{:}T.t$	抽象
	$t\ t$	应用
	$\lambda X.t$	类型抽象
	$t\ [T]$	类型应用
v ::=		值：
	$\lambda x{:}T.t$	抽象值
	$\lambda X.t$	类型抽象值
T ::=		类型：
	X	类型变量
	$T \rightarrow T$	函数的类型
	$\forall X.T$	全称类型
Γ ::=		上下文：
	$\varnothing$	空上下文
	$\Gamma, x{:}T$	项变量绑定
	Γ, X	类型变量绑定

求值 $\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-APP1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(E-APP2)}$$

$$(\lambda x{:}T_{11}.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \quad \text{(E-APPABS)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ [T_2] \longrightarrow t_1'\ [T_2]} \quad \text{(E-TAPP)}$$

$$(\lambda X.t_{12})\ [T_2] \longrightarrow [X \mapsto T_2]t_{12} \quad \text{(E-TAPPTABS)}$$

类型 $\boxed{\Gamma \vdash t : T}$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}$$

$$\frac{\Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.t_2 : T_1 \rightarrow T_2} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{(T-APP)}$$

$$\frac{\Gamma, X \vdash t_2 : T_2}{\Gamma \vdash \lambda X.t_2 : \forall X.T_2} \quad \text{(T-TABS)}$$

$$\frac{\Gamma \vdash t_1 : \forall X.T_{12}}{\Gamma \vdash t_1\ [T_2] : [X \mapsto T_2]T_{12}} \quad \text{(T-TAPP)}$$

图 23.1　多态 lambda 演算(系统 F)

热身例子

我们已经知道类型抽象和应用是怎样用来定义单一的多态恒等函数：

```
id = λX. λx:X. x;
▸ id : ∀X. X → X
```

并将其实例化，产生任何满足要求的具体恒等函数：

```
id [Nat];
▸ <fun> : Nat → Nat

id [Nat] 0;
▸ 0 : Nat
```

一个更有用的例子是多态加倍函数：

```
double = λX. λf:X→X. λa:X. f (f a);
▸ double : ∀X. (X→X) → X → X
```

类型 X 的抽象使我们用不同类型参数实例化 double，从而获得有具体类型的 doubling(加倍)函数：

```
doubleNat = double [Nat];
▸ doubleNat : (Nat→Nat) → Nat → Nat

doubleNatArrowNat = double [Nat→Nat];
▸ doubleNatArrowNat : ((Nat→Nat)→Nat→Nat) →
                      (Nat→Nat) → Nat → Nat
```

一旦用类型参数实例化了，double 能进一步被用在实际的函数和合适的类型参数上：

```
double [Nat] (λx:Nat. succ(succ(x))) 3;
▸ 7 : Nat
```

这里有一个稍难的例子——多态形的自应用。可回忆，在前面介绍的简单类型 lambda 演算中，没有办法给无类型项 λx.x x 加上类型(参见练习 9.3.2)。另一方面，在系统 F 中，只要我们给 x 一个多态类型并将该项合理地实例化，则该项就成为可类型化的：

```
selfApp = λx:∀X.X→X.  x [∀X.X→X] x;
▸ selfApp : (∀X. X→X) → (∀X. X → X)
```

作为一个更有用的自应用例子，我们还能将多态的 double 函数应用于自身，可产生一个多态四倍函数：

```
quadruple = λX. double [X→X] (double [X]);
▸ quadruple : ∀X. (X→X) → X → X
```

23.4.1　练习[★ ↛]：使用图 23.1 中的类型规则，证明一下上面的项给出的类型。

多态列表

在大多数现实世界中带多态的编程比上面举的例子要简单得多。这里举一个直接多态编程的例子，假设编程语言带类型构造子 List，以及通常列表操作原语的项构造子，类型如下：

```
▸ nil : ∀X. List X
  cons : ∀X. X → List X → List X
  isnil : ∀X. List X → Bool
  head : ∀X. List X → X
  tail : ∀X. List X → List X
```

当第一次在 11.12 节中引入列表时，采用“习惯”的推论规则，允许对带任意类型元素的列表进行操作。这里，我们给操作加上多态类型，用来表达完全相同的约束，即列表不再需要植入到程序的核心中，可以简单地认为是提供几种带特殊多态类型的常量库。我们还可对第 13 章中介绍的 Ref 类型和引用单元的原语操作，以及许多其他的通用数据和控制结构，按照对列表相同的方法来处理。

可以用这些原语来对 List 定义自己的多态操作。例如，这里有个多态 map 函数：使用将 X 映射到 Y 的函数和 X 的列表，返回 Y 列表：

```
map = λX. λY.
        λf: X→Y.
          (fix (λm: (List X) → (List Y).
                  λl: List X.
                    if isnil [X] l
                      then nil [Y]
                      else cons [Y] (f (head [X] l))
                                    (m (tail [X] l))));
▸ map : ∀X. ∀Y. (X→Y) → List X → List Y

  l = cons [Nat] 4 (cons [Nat] 3 (cons [Nat] 2 (nil [Nat])));
▸ l : List Nat

  head [Nat] (map [Nat] [Nat] (λx:Nat. succ x) l);
▸ 5 : Nat
```

23.4.2 练习[★ ↛]:请证明一下上面的 map 确实具有所示的类型。

23.4.3 练习[推荐,★★]:请模仿 map,写出一个多态列表倒序的函数:

```
reverse : ∀X. List X → List X.
```

这道题最好上机完成。用 fullomega 检查器,并从 fullomega 目录中拷贝 test.f 的内容到自己的输入文件的顶端[该文件含有 List 构造子的定义及一些相关要求使用 F_ω 系统(第 29 章中有介绍)功能强大的抽象工具的操作。不需要去理解它们是怎样具体完成本练习的]。

23.4.4 练习[★★ ↛]:写出一个简单的多态排序函数:

```
sort : ∀X. (X→X→Bool) → (List X) → List X
```

其中第一个参数是类型 X 的元素的比较函数。

Church 编码

在 5.2 节中,我们知道许多原始类型,如布尔型、数字型和列表型都能在纯无类型 lambda 演算中作为函数进行编码。在本节中,还要说明这些 Church 编码是怎样在系统 F 中运行的。读者可以参考 5.2 节的内容,更新对 Church 编码的印象。

该编码有两点是很有趣的。其一,它们能使我们很好地理解类型抽象和应用;其二,它们说明系统 F 与纯无类型 lambda 演算一样,是一种用于计算的非常丰富的语言,因为纯系统能表达大量的数据和控制结构。其意思是说,如果我们后来设计一个全面的编程语言,将系统 F 作为其核心,就可将这些特点作为原语(这样是为了提高效率,使我们能引入更方便具体的语法)而不需要改动核心语言的任何基本属性。当然,不需要改动任何基本属性这种说法只对部分高级语言来说是正确的。例如,增加一个引用到系统 F 中,做法和第 13 章的 $\lambda_{\leftarrow}$ 一样,就是对系统中基本计算特性的一个实际改动。

让我们从 Church 布尔型开始。回忆一下无类型的 lambda 演算,用 lambda 项 tru 和 fls 来表示布尔常量 true 和 false,如下所示:

```
tru = λt. λf. t;
fls = λt. λf. f;
```

这里的每一项都使用两个参数并返回其中一个。如果想给 tru 和 fls 分配一个通用的类型,最好假设这两个参数具有相同的类型(调用者并不知道是哪一个与 tru 或 fls 关联),但这种

类型可以是任意的(因为 tru 和 fls 不需要对它们的参数做任何事情,只是要返回其中一个)。这样,就得到了 tru 和 fls 的如下类型:

```
CBool = ∀X.X→X→X;
```

这样,只要加进了合适的类型注释给上面的无类型形式,就得到了系统 F 的项 tru 和 fls:

```
  tru = λX. λt:X. λf:X. t;
▸ tru : CBool

  fls = λX. λt:X. λf:X. f;
▸ fls : CBool
```

可以通过构造一个新的布尔型,即用一个存在的布尔类型来决定该返回哪一个参数,以写一个通用的布尔型操作,如 not:

```
  not = λb:CBool. λX. λt:X. λf:X. b [X] f t;
▸ not : CBool → CBool
```

23.4.5 练习[推荐,★]:写一个项 and,使用两个 CBool 类型的参数并计算它们的合取式。

我们也能给数字写一个类似的程序。5.2 节中介绍的 Church 数子对每个自然数 n 编一段函数代码,使用两个参数 s 和 z,并将 s 用于 z,如此重复 n 次:

```
c₀ = λs. λz. z;
c₁ = λs. λz. s z;
c₂ = λs. λz. s (s z);
c₃ = λs. λz. s (s (s z));
```

显然,z 应该与 s 的定义域类型相同,且 s 返回的结果也应该有相同的类型。由此,我们得出系统 F 中 Church 数子的类型为:

```
CNat = ∀X. (X→X) → X → X;
```

若给无类型的 Church 数字加上合适的注释,就能得到该类型的元素:

```
  c₀  =  λX. λs:X→X. λz:X. z;
▸ c₀ : CNat

  c₁  =  λX. λs:X→X. λz:X. s z;
▸ c₁ : CNat

  c₂  =  λX. λs:X→X. λz:X. s (s z);
▸ c₂ : CNat
```

关于 Church 数子的有类型后继(successor)函数可以定义成为:

```
  csucc = λn:CNat. λX. λs:X→X. λz:X. s (n [X] s z);
▸ csucc : CNat → CNat
```

即对给定的 s 和 z,csucc n 返回一个 CNat 的元素,将 s 用到 z 上 n 次(即将 n 应用于 s 和 z),然后再重复一次。其他的算术操作也可以类似地定义。例如,加法操作也能在后继函数中定义,

```
cplus = λm:CNat. λn:CNat. m [CNat] csucc n;
▸ cplus : CNat → CNat → CNat
```

或更直接地:

```
cplus = λm:CNat. λn:CNat.  λX. λs:X→X. λz:X. m [X] s (n [X] s
▸ cplus : CNat → CNat → CNat
```

如果语言也包括原始数字(参见图 8.2),那么也用下列的函数将 Church 数子转化为一般的数字:

```
cnat2nat = λm:CNat. m [Nat] (λx:Nat. succ(x)) 0;
▸ cnat2nat : CNat → Nat
```

这样就可以相信我们对 Church 数字的操作确实是计算了期望的算术函数:

```
cnat2nat (cplus (csucc c0) (csucc (csucc c0)));
▸ 3 : Nat
```

23.4.6 练习[推荐,★★]:请写一个 iszero 函数,使其用于 Church 数字 c_0 和 fls 等时,能返回 tru。

23.4.7 练习[★★ ↛]:请证实下列的项:

```
ctimes = λm:CNat. λn:CNat.  λX. λs:X→X. n [X] (m [X] s);
▸ ctimes : CNat → CNat → CNat

cexp = λm:CNat. λn:CNat.  λX. n [X→X] (m [X]);
▸ cexp : CNat → CNat → CNat
```

为有指示的类型。请给出一个正式的论述,说明它们能实现算术的乘法和求幂操作。

23.4.8 练习[推荐,★★]:请说明类型:

```
PairNat = ∀X. (CNat→CNat→X) → X;
```

能表示多个数定序对,其中要写出下列函数:

```
pairNat : CNat→CNat→PairNat;
fstNat : PairNat→CNat;
sndNat : PairNat→CNat;
```

用来构造来自二元数字对的类型元素,并能存取它们的第一组成部分和第二组成部分。

23.4.9 练习[推荐,★★★]:用练习 23.4.8 中定义的函数写一个计算 Church 数子前驱的函数 pred(若输入为 0 则返回 0)。提示:有关思想在 5.2 节有所提到。定义一个函数 f: PairNat→PairNat,将序对(i,j)映射为$(i+1,i)$,也就是说,将序对中的第二个部分去掉,将第一部分复制给第二部分,然后给第一部分加 1。这样从(0,0)开始实现 n 次,能产生$(n,n-1)$,并从中提取出第二部分就得出了 n 的前驱。

23.4.10 练习[★★★]:还有另一种计算 Church 数子的前驱函数,以 k 表示无类型 lambda 项 λx.λy.x 和以 i 表示 λx.x. 以下无类型 lambda 项:

```
vpred = λn. λs. λz. n (λp. λq. q (p s)) (k z) i
```

(该式来自于 Barendregt,1992,他认为该式是引用 J.Velmans 的)可以计算无类型 Church 数子的前驱。请说明,在系统 F 中,如果给该项增加类型抽象和应用,并用合适的类型给无类型项注释囿变量,则该项能类型化。为了更有说服力,请解释一下它是怎样得出来的。

编码列表

作为最后的例子,我们要将 Church 编码从数字扩充到列表。对表达有力的纯系统 F 而言,这是精彩的示范,因为它能说明所有以上介绍过的关于多态列表操作的编程实例都能实际地在纯语言中表达出来(为了方便,我们确实用 fix 结构来定义一般的递归函数,但本质上说,不用它,同样的结构也能被实现,参见练习 23.4.11 和练习 23.4.12.)。

在练习 5.2.8 中见到列表也能用类似于自然数字编码的方式在无类型 lambda 演算中编码。有效地,在一元符号中,一个数字像一个虚元素的列表。若将这种思想推广到任意类型的元素中,就能得到列表的 Church 编码形式,它将一个有元素 x,y 和 z 的列表表示成一个函数,只要给定任意函数 f 和始值 v,就能计算 f x (f y (f z v))。在 OCaml 术语中,列表是用它的函数 fold_right 来表示的。

带类型 X 元素的列表类型 List X 的定义如下:

```
List X = ∀R. (X→R→R) → R → R;
```

在该列表表示下,nil 值很容易写出[①]:

```
nil = λX. (λR. λc:X→R→R. λn:R. n) as List X;
▸ nil : ∀X. List X
```

cons 和 isnil 操作也容易表示:

```
cons = λX. λhd:X. λtl:List X.
           (λR. λc:X→R→R. λn:R. c hd (tl [R] c n)) as List X;
▸ cons : ∀X. X → List X → List X

isnil = λX. λl:List X. l [Bool] (λhd:X. λtl:Bool. false) true;
▸ isnil : ∀X. List X → Bool
```

至于 head 操作,有些麻烦。第一个难点是该怎样处理空列表的头节点。可以回想一下,如果在语言中有不动点运算符,就可以用它来构建任意类型的表达式。事实上,利用类型抽象,就能更深入,写一个单独的合一函数,对给出的类型 X 将其应用于 unit 时,会产生一个从 Unit 到 X 的发散函数:

```
diverge = λX. λ_:Unit. fix (λx:X. x);
▸ diverge : ∀X. Unit → X
```

这样,我们就能将 diverge[X]unit 作为 head[X]nil 的结果。

```
head = λX. λl:List X. l [X] (λhd:X. λtl:X. hd) (diverge [X] unit);
▸ head : ∀X. List X → X
```

糟糕的是,该定义还不是我们真正需要的:它总是会发散,即使是用在一个非空列表中。

① 这里的 as 注释可帮助类型检查器以可读的方式打印出 nil 的类型。如在 11.4 节中所见,本书中所有的类型检查器在打印类型之前要实现简单的缩写压缩,但压缩函数不能灵活到能自动处理"参数缩写",如 List 的地步。

为得到令人满意的定义,需要将其稍微修改一下,将 diverge[X]作为参数赋给 l 时,不必传递它的 Unit 参数。为做到这一点,只要删掉 unit 参数,并相应地将第一个参数的类型改为 l:

```
head =
  λX. λl:List X.
    (l [Unit→X] (λhd:X. λtl:Unit→X. λ_:Unit. hd) (diverge [X]))
    unit;
▸ head : ∀X. List X → X
```

即,l 用于类型 X →(Unit →X)→(Unit →X)的函数和类型 Unit →X 的基础值,它创建了类型为 Unit→X 的函数。如果 l 表示的是空列表,该结果将变成 diverge[X];但如果 l 表示的是非空列表,结果将是一个利用 unit 返回 l 头元素的函数。最后 l 的结果可用于 unit 去获得实际的头元素(或没那么走运,是发散的),这样就得到了期望的 head。

对 tail 函数,我们对 Church 编码的序对(第一部分为类型 X,第二部分为类型 Y)采用缩写形式 Pair X Y(根据练习 23.4.8,将 PairNat 型进行一般化):

```
Pair X Y = ∀R. (X→Y→R) → R;
```

对它的操作也只是简单地对类型 PairNat 一般化过程:

```
▸ pair : ∀X. ∀Y. X → Y → Pair X Y
  fst : ∀X. ∀Y. Pair X Y → X
  snd : ∀X. ∀Y. Pair X Y → Y
```

现在 tail 函数可写成:

```
tail =
  λX. λl: List X.
    (fst [List X] [List X] (
       l [Pair (List X) (List X)]
         (λhd: X. λtl: Pair (List X) (List X).
           pair [List X] [List X]
             (snd [List X] [List X] tl)
             (cons [X] hd (snd [List X] [List X] tl)))
         (pair [List X] [List X] (nil [X]) (nil [X]))));
▸ tail : ∀X. List X → List X
```

23.4.11 练习[推荐,★★]:严格地说,本小节中的例子并不是在纯系统 F 中表达的,因为当 head 用于空列表时,我们使用了一个 fix 算子来构建一个返回值。请将 head 修改一下,使其在列表为空时能返回一个特别的参数(而不是发散的)。

23.4.12 练习[推荐,★★★]:在纯系统 F 中(无 fix),写出一个如下类型的 insert 函数:

```
∀X. (X→X→Bool) → List X → X → List X
```

调用一个比较函数,一个排序列表和一个新元素,并将该元素插入到列表的适当位置(即放在所有比它小的元素的后面)。接着,再用 insert 建立一个纯系统 F 中的列表排序函数。

23.5 基本性质

系统 F 的基本特性与简单类型 lambda 演算的特性十分相似。尤其是,类型保持与进展证明都是对第 9 章中介绍的直接扩充。

23.5.1 定理[保留]:如果 $\Gamma \vdash t:T$ 且 $t \to t'$,则 $\Gamma \vdash t':T$。

证明:作为练习[推荐,★★★]。

23.5.2 定理[进展]:若 t 是一个封闭的良类型的项,则 t 是一个值,否则存在某 t′使 $t \to t'$。

证明:作为练习[推荐,★★★]。

系统 F 与 $\lambda_{\to}$ 一样有规范化特性——作为每个良类型程序求值可终止[①]。与上面的类型安全定理不同,规范化是非常难证明的(的确,如在练习 23.4.12 中所做的,我们能在纯语言中编写如排序函数等这样的代码,而不必用到 fix,这确实是十分惊人的)。该证明,以第 12 章中一般化方法为基础,也是 Girard 的博士论文的主要部分(1972; 还可参见 Girard, Lafont 和 Taylor, 1989)。从那时候起,他的证明方法被许多人分析并改进;参见 Gallier(1990)。

23.5.3 定理[规范化]:良类型的系统 F 项是规范化的。

23.6 抹除,可类型化,类型重构

如我们在 9.5 节所做的,可定义一个类型抹除函数,该函数通过抹掉所有系统 F 项和无类型 lambda 项中的类型注释(包括所有的类型抽象和应用),来建立起系统 F 项到无类型 lambda 项之间的映射:

$$
\begin{array}{lcl}
erase(\mathtt{x}) & = & \mathtt{x} \\
erase(\lambda \mathtt{x}{:}\mathtt{T_1}.\ \mathtt{t_2}) & = & \lambda \mathtt{x}.\ erase(\mathtt{t_2}) \\
erase(\mathtt{t_1}\ \mathtt{t_2}) & = & erase(\mathtt{t_1})\ erase(\mathtt{t_2}) \\
erase(\lambda \mathtt{X}.\ \mathtt{t_2}) & = & erase(\mathtt{t_2}) \\
erase(\mathtt{t_1}\ [\mathtt{T_2}]) & = & erase(\mathtt{t_1})
\end{array}
$$

如果存在某良类型的项 t 满足 $erase(t) = m$,则无类型 lambda 演算中的项 M 在系统 F 中是可类型化的。对给定的无类型项,类型重构留给我们的问题是,能否找到某良类型能抹除类型为 m 的项。

系统 F 的类型重构是编程语言中时间最长的,难以解决的问题之一,该难题从 20 世纪 70 年代的早期一直遗留,直至 20 世纪 90 年代初被 Wells(否定地)解决。

23.6.1 定理[WELLS,1994]:已知无类型 lambda 演算的封闭项 m,系统 F 中是否存在良类型的项 t 使 erase(t) = m 是无法确定的。

不但是全类型重构,而且部分类型重构的多种形式在系统 F 中也是不可确定的。例如,以下"部分抹除"函数,除了参数进行了类型应用,它都会保持所有类型注释的完整性:

$$
\begin{array}{lcl}
erase_p(\mathtt{x}) & = & \mathtt{x} \\
erase_p(\lambda \mathtt{x}{:}\mathtt{T_1}.\ \mathtt{t_2}) & = & \lambda \mathtt{x}{:}\mathtt{T_1}.\ erase_p(\mathtt{t_2}) \\
erase_p(\mathtt{t_1}\ \mathtt{t_2}) & = & erase_p(\mathtt{t_1})\ erase_p(\mathtt{t_2}) \\
erase_p(\lambda \mathtt{X}.\ \mathtt{t_2}) & = & \lambda \mathtt{X}.\ erase_p(\mathtt{t_2}) \\
erase_p(\mathtt{t_1}\ [\mathtt{T_2}]) & = & erase_p(\mathtt{t_1})\ [\,]
\end{array}
$$

① 的确,基于全 β 归约且可带更多可操作性语义的系统 F 形式具有强规范化特性:每个从良类型项开始的归约路径能保证有终点。

注意,在抹除项中仍然保持类型应用的记号(空括号),所以我们知道它们必须在哪里出现,只是用的是什么样的类型还不知道。

23.6.2 定理[Boehm 1985,1989]:对一个有类型应用标志但忽略了参数的封闭项 s,是否存在某良类型系统 F 中的项 t 满足 $erase_p(t) = s$ 是无法确定的。

Boehm 说明了这种类型重构的形式与高阶合一化同样困难,因此也是无法确定的。有趣地很,这种否定的结果竟然能直接得出一个有用的部分类型重构技术(Pfenning,1988,1993a),这一点是基于 Huet 的早期对高阶合一化的有效准算法(Huet,1975)的研究成果。以后对该成果的改进包括对高阶约束求解的更精确算法(Dowek,Hardin,Kirchner 和 Pfenning,1996),该算法减少了无终止和产生非惟一解可能性的发生。其他一些相关算法如 LEAP(Pfenning 和 Lee,1991),Elf(Pfenning,1989)和 FX(O'Toole 和 Gifford,1989)在实践中都效果不错。

Perry 的著作中就部分类型重构提出了不同的方法,提出一阶存在类型(参见第 24 章)可以溶入到 ML 的数据类型(datatype)机制中(Perry,1990);该思想又被 Läufer 和 Odersky (Läufer,1992; Läufer 和 Odersky,1994)深入研究。本质上,数据类型的构造器和析构器可以认为是明确的类型注释,它会说明哪里需要插入值,哪里的值是从拆开的联合类型投影过来的,哪里的递归类型必须隐藏或显露,以及(如果加入了存在类型)哪里应该打包或哪里应该解包,等等。该思想被扩充,将 Rémy(1994)提出的一级(不可预言的)全称量词包含进来。更近的提议是由 Odersky 和 Läufer(1996)提出的,又由 Garrigue 和 Rémy (1997)进行了深入研究,他们适当地扩充了 ML 型的类型重构,允许编程人员明确地用类型注释函数的参数,因为该参数(不同于那些可以自动推断的注释)能包含嵌入的全称量词,从而部分地造成 ML 与更强大的不可预言系统之间的差距。关于类型重构的方法族的优点是它相对简单且简洁地与 ML 的多态成为一体。

Pierce 和 Turner(1998)提出了解决系统部分类型重构的一个实用的方法,称为局部类型推论(或局部类型重构),该方法包括了子类型和不可预言多态(还可参见 Pierce 和 Turner,1997; Hosoya 和 Pierce,1999)。局部类型推论已经在几种近期的语言设计中出现,包括 GJ(Bracha,Odersky,Stoutamire 和 Wadler,1998)和 Funnel(Odersky 和 Zenger,2001),后者介绍了一个更强大的形式——有色局部类型推论(Odersky,Zenger 和 Zenger,2001)。

一个更简单但可预言性小的贪婪类型推论算法由 Cardelli(1993)提出了;还有些类似的算法已经用于独立类型理论的证明检查器中,如 NuPrl(Howe,1988)和 Lego (Pollack,1990)。这里的思想是任何类型的类型注释都有可能被程序员忽略:分析器会为每种类型注释产生一个新的合一变量 X。在类型检查过程中,子类型检查算法会用来检查是否某些类型 S 是子类型 T,这里 S 和 T 都包含了合一变量。子类型检查过程通常是达到满足 X <:T 或 T <:X 这个子目标为止,这时点 X 被实例化为 T,这样以一种最简单可行的方式满足直接约束。然而,将 X 设置为 T 并不是最好的选择,这样还会造成后来对含 X 类型的子类型检查失败,若换一个选择还有可能使子类型检查成功;但 Cardelli 的实现和早期的 Pict 语言(Pierce 和 Turner,2000)的实践都表明该算法的贪婪选择在几乎所有的情形之下都是正确的。不过,一旦贪婪算法运行出错,它的行为会使程序员十分迷惑,因为会产生一些难理解的错误,远远达不到产生次优实例的结果。

23.6.3 练习[★★★★]:规范化特性表明无类型项 omega = (λx.x x)(λy.y y)不能在系统 F 中类型化,因为 omega 的归约不能达到一个标准的形式。然而,对这个情况可以用一个更直

接的"组合"证明方式，只要利用定义类型关系的规则即可。

1. 让我们调用一个系统 F 的项 exposed，如果它是一个变量，一个抽象型 $\lambda x{:}T.t$，或一个应用 t s（也就是说如果它不是一个类型抽象 $\lambda X.t$ 或类型应用 $t[S]$）。

   请说明如果（在某些上下文中）t 是良类型的且 *erase*(t) = m，则存在某 exposed 项 s，满足 *erase*(s) = m 且 s 是良类型的（可能在不同的上下文中）。

2. 将类型抽象嵌套序列 $\lambda X_1 \cdots \lambda X_n.t$ 简写为 $\lambda \overline{X}.t$。类似地，将类型应用嵌套序列 $((t[A_1])\cdots[A_{n-1}])[A_n]$ 简写为 $t[\overline{A}]$，将多态类型序列 $\forall X_1 \cdots \forall X_n.T$ 简写为 $\forall \overline{X}.T$。注意，这种的序列允许为空。例如，如果是一个类型变量的空序列，则 $\forall \overline{X}.T$ 就只含 T。

   请说明如果 *erase*(*t*) = m 及 $\Gamma \vdash t{:}T$，存在形为 $\lambda \overline{X}.(u\ [\overline{A}])$ 的某项 s，其中 $\overline{X}$ 为某类型变量序列，$\overline{A}$ 为某类型序列，u 为 exposed 项，使下式成立：*erase*(s) = m 且 $\Gamma \vdash s{:}T$。

3. 请说明，如果 t 是一个类型 T 的（上下文为 Γ）exposed 项且 *erase*(t) = m n，则 t 有形式 s u，其中 s 和 u 均为项，满足 *erase*(s) = m 且 *erase*(u) = n，且 $\Gamma \vdash s{:}\ U \rightarrow T$ 和 $\Gamma \vdash u{:}U$。

4. 设 $x{:}T \in \Gamma$。请说明如果 $\Gamma \vdash u{:}\ U$ 且 *erase*(u) = x x，那么：

   (a) $T = \forall \overline{X}.X_i$ 其中 $X_i \in \overline{X}$ 成立，

   或者：

   (b) $T = \forall \overline{X_1}\,\overline{X_2}.T_1 \rightarrow T_2$ 成立，其中对类型序列 $\overline{A}$ 和 $\overline{B}$ 有 $|\overline{A}| = |\overline{X_1}\,\overline{X_2}|$ 和 $|\overline{B}| = |\overline{X_1}|$，满足 $[\overline{X_1}\,\overline{X_2} \mapsto A]T_1 = [\overline{X_1} \mapsto B](\forall \overline{Z}.T_1 \rightarrow T_2)$

5. 请说明如果 *erase*(s) = $\lambda x.m$ 和 $\Gamma \vdash s{:}S$，那么对某 $\overline{X}, S_1$ 和 S_2，则 S 有形式 $\forall \overline{X}.S_1 \rightarrow S_2$。

6. 定义类型 T 的形式 *leftmost leaf* 如下：

   $$
   \begin{aligned}
   \textit{leftmost-leaf}(X) &= X\\
   \textit{leftmost-leaf}(S \rightarrow T) &= \textit{leftmost-leaf}(S)\\
   \textit{leftmost-leaf}(\forall X.S) &= \textit{leftmost-leaf}(S).
   \end{aligned}
   $$

   请说明，如果 $[\overline{X_1}\,\overline{X_2} \mapsto \overline{A}](\forall \overline{Y}.T_1) = [\overline{X_1} \mapsto \overline{B}](\forall \overline{Z}.(\forall \overline{Y}.T_1) \rightarrow T_2)$，则对某些 $X_i \in \overline{X_1}\,\overline{X_2}$，必有 *leftmost-leaf*$(T_1) = X_i$ 成立。

7. 请说明 omega 在系统 F 中是不可类型化的。

23.7 抹除和求值顺序

图 23.1 中赋予系统 F 的操作语义是一个类型传递语义：当一个多态函数遇到一个类型参数时，类型被代换入函数体内。在第 25 章中系统 F 的 ML 实现过程就是这样。

在一个基于系统 F 的一个更现实的编程语言的翻译器或编译器中，运行时对类型的处理损耗巨大。更有甚者，类型注释在运行时起不到重要作用，因为在运行时程序做的决定并不是基于类型的：我们可以试着拿一个良类型程序，用任意的方式重写它的类型注释，然后得到同样处理方式的程序。因此，许多多态语言改为采用类型抹除语义，检查完了词语后，所有的类型都被抹除且产生的无类型项会被解释或编译成机器代码①。

---

① 在一些语言中，有些特性，如 cast（参见 15.5 节）会强制类型传递的实现。这种语言的高性能的实现主要是为了保持运行时类型信息的初始形式，即只在实际需要使用时才将类型传给多态函数。

然而,即使是在一个成熟的语言中,也免不了一些副作用的特点,如易变的引用单元或异常,类型抹除函数还需要比 23.6 节中提到的全抹除函数更细致的定义。例如,如果我们用一个异常提升原语 error(参见 14.1 节)来扩充系统 F,则项:

```
let f = (λX.error) in 0;
```

求值为 0,因为 λX.error 是一个语法值,其中 error 从来没有求值过,而它的抹除:

```
let f = error in 0;
```

当求值时会出现一个异常[①]。这表明了类型抽象确实起到一个重要的语义作用,因为它们在值调用求值方式下会停止求值,这样就能拖延或防止副作用求值的发生。

我们能通过引进一个新的适合值调用求值的抹除形式来修补这个差距,这里将类型抽象抹除为项抽象:

$$
\begin{aligned}
erase_v(\mathtt{x}) &= \mathtt{x} \\
erase_v(\lambda \mathtt{x}:\mathtt{T}_1.\ \mathtt{t}_2) &= \lambda \mathtt{x}.\ erase_v(\mathtt{t}_2) \\
erase_v(\mathtt{t}_1\ \mathtt{t}_2) &= erase_v(\mathtt{t}_1)\ erase_v(\mathtt{t}_2) \\
erase_v(\lambda \mathtt{X}.\ \mathtt{t}_2) &= \lambda_.\ erase_v(\mathtt{t}_2) \\
erase_v(\mathtt{t}_1\ [\mathtt{T}_2]) &= erase_v(\mathtt{t}_1)\ \mathtt{dummyv}
\end{aligned}
$$

这里 dummyv 是一个和 unit 一样的任意的无类型值[②]。该新抹除函数的合理之处是它能与无类型求值交换,也就是说抹除和求值可调换顺序进行。

23.7.2 定理:如果 $erase_v(t) = u$,则(1)根据 t 和 u 的各自的求值关系,它们都是范式;(2)$t \to t'$ 和 $u \to u'$ 有 $erase_v(t') = u'$ 成立。

23.8 系统 F 片断

精确且功能强大的系统 F 已经在多态理论研究领域占据了一个中心地位。但是,对语言设计来说,类型重构带来的损耗有时太大而不能作为一个特性,因为它的全部功能很少用到。于是出现了许多带更多易处理类型重构问题的系统 F 受限片断的提议。

其中最受欢迎的是 ML 的 let 多态(参见 22.7 节),有时它也可称为前束多态,因为它可被视为类型变量范围只限于无量词类型(单一类型)的系统 F 片断,而且该系统中量化类型(多态类型或类型方案)不允许出现在箭头的左端。ML 中 let 的特殊作用使得对应难以精确描述;详见 Jim(1995)。

另一种研究得较好的系统 F 的限制是 2 秩多态,先由 Leivant(1983)提出并由许多其他的学者进一步研究(参见 Jim,1995,1996)。当一个类型以树的形式画出时,如果没有一条从类型的根部到任意一个量词的路径经过 2 或更多箭头的左端,则称该类型为 2 秩。例如,$(\forall X.X \to X) \to \mathrm{Nat}$ 是 2 秩的,$\mathrm{Nat} \to \mathrm{Nat}$ 和 $\mathrm{Nat} \to (\forall X.X \to X) \to \mathrm{Nat} \to \mathrm{Nat}$ 也是 2 秩的,但$((\forall X.X \to X) \to$

① 这与我们在 22.7 节中见到的不合理的参照与 ML 型 let 多态的组合的问题有关系。其中在该例子中的 let 体的一般化与这里的明确的类型抽象对应。

② 相比之下,为恢复在 22.7 节中出现了副作用的 ML 型类型重构的可靠性而加入的值限制的确起到了抹除类型抽象的作用(一般化一个类型变量是与抹除一个类型抽象相反的过程),但确保了可靠性,它允许这种一般化只在推断的类型抽象的发生即将接近一个项抽象或其他的语法值构造子时进行一般化,而且这样还能停止求值。

Nat)→Nat 不是。在2秩系统中,所有的类型都必须是2秩的。这种系统比前束(ML)片断功能更强大,因为它能将类型指派给更多的无类型 lambda 项。

23.7.1 练习[推荐,★★]:用图13.1的引用将22.7节中的不合理例子(即练习22.7.1前面举的那个例子)扩充为系统F。

Kfoury 和 Tiuryn(1990)曾证明系统F的2秩片断类型重构的复杂度与ML系统相同(即Dexptime完全的)。Kfoury 和 Wells(1999)给2秩系统第一个正确的类型重构算法,并说明系统F的3秩或更高秩的类型重构是不可确定的。

除了量词外,2秩限制还可用在其他功能强大的类型重构中。例如,除了出现在2或更多箭头左端的交叉类型外,交叉类型(参见15.7节)都是2秩的(Kfoury, Mairson, Turbak 和 Wells, 1999)。系统F的2秩片断和一阶交叉类型系统的2秩片断是紧密联系在一起的。的确,Jim(1995)曾说明过它们能给同一个无类型项准确类型化。

23.9 参数性

回想一下23.4节我们是怎样定义 Church 布尔类型 CBool 的:

```
CBool = ∀X.X→X→X;
```

以及常量 tru 和 fls:

```
  tru = λX. λt:X. λf:X. t;
▸ tru : CBool
  fls = λX. λt:X. λf:X. f;
▸ fls : CBool
```

只要有了类型 CBool,我们简单地根据类型的结构就可机械地写出 tru 和 fls 的定义。因为 CBool 是从符号∀开始的,则类型 CBool 的任何值都将是个类型抽象,所以 tru 和 fls 必须以 λX 开始。因为 CBool 体中是一个箭头类型 X→X→X,该类型的每个值必须取用类型 X 的两个参数,即 tru 和 fls 体都必须以 λt:X.λf:X 开始。最后,因为 CBool 的结果类型是 X,所以类型 CBool 的任何值都应为类型 X 的元素。但 X 是一个参数,可能返回的该类型的惟一值是囿变量 t 和 f——我们没有别的获得或构建该类型值的方法。换个方式说,tru 和 fls 基本上是类型 CBool 的惟一宿主。严格地说,CBool 包含了一些其他的项,如(λb:CBool.b) tru,但很显然,它们中的任何一项的实现都与 tru 或 fls 相同。

上面的现象是一个称为参数性规则的结果,该规则形式化了多态程序的合一行为。参数性是由 Reynolds(1974,1983)提出的,随同相关的一些概念,它又由一些学者进一步研究,这些人是 Reynolds (1984, Reynolds 和 Plotkin, 1993), Bainbridge 等(1990), Ma(1992), Mitchell(1986), Mithchell 和 Meyer(1985), Hasegawa(1991), Pitts(1987, 1989, 2000), Abadi, Cardelli, Curien 和 Plotkin (Abadi, Cardelli 和 Curien, 1993; Plotkin 和 Abadi, 1993; Plotkin, Abadi 和 Cardelli, 1994), Wadler (1989, 2001),等等。参见 Wadler (1989)的说明性介绍。

23.10 不可预言性

系统 F 的多态经常被称为不可预言的。通常,集合、类型等的定义如果包括了一个量词(它的定义域包括了正在定义的事物)都被称为不可预言的。例如,在系统 F 中,类型 $T = \forall X.X \to X$ 中的类型变量 X 的范围是所有的类型,包括 T 本身(所以我们也能用类型 T 来实例化 T 的项,产生一个从 T 到 T 的函数)。另一方面,ML 中的多态常被称为可预言的(或分层的),因为类型变量的范围是单一类型,该类型不含量词。

术语"可预言的"和"不可预言的"都出自逻辑学。Quine(1987)用简短而清晰的话说明了一下它们的历史:

在与 Henri Poincaré 的交流中……Russell 暂时将他的矛盾归功于他称为恶性循环的谬误。该"谬误"是用一个成员条件来指定一个类,且这个成员条件是直接或间接引用大量的类,而这些类中有一个就是正在被指定的类。比如 Russell 矛盾中的成员条件就是非自我成员:x 不是 x 的成员,该矛盾使得成员条件下的 x,在其他的情况下,就是正由成员条件定义的那个类。Russell 和 Poincaré 称这种成员条件为不可预言性,意思是不能指定一个类。于是,Russell 和其他人的集合理论中的矛盾就消去了……

术语,"可预言"和"不可预言"又是从何而来的? 在 Russell 的术语中,有关类和成员条件的老掉牙的说法是每个谓词决定了一个类;于是他从那些从来不会用于决定类的成员条件中提取出谓词的标题拼凑了这个老词。"可预言"不是指特别的分层方法,也不是暗指什么构建的过程;它只是 Russell 和 Poincaré 特别提到该接受什么样的成员条件作为类的产物或"可预言"。但很快,事情走向了反面,现在可预言集合理论已称为构造集合理论,不可预言的定义已在前面解释过了,而不管该选什么样的成员条件来决定类。

23.11 注释

若想深入了解系统 F,可参阅 Reynold 的介绍性文章(1990)和他的《编程语言理论》[Theories of Programming Languages(1998b)]一书。

第24章 存在类型[1]

在类型系统中检查了全称量词的作用后(参见第23章),很自然会想到是否存在量词在编程中也是十分有用的。的确如此,它们为数据抽象和信息隐藏提供了很好的基础。

24.1 引言

存在类型基本上与全称类型一样复杂(事实上,将在24.3节中看到存在类型可根据全称直接编码)。然而,存在类型的产生和消去形式,在语法上比与全称有关的简单类型抽象和运用都要更难,有些人从一开始就会感到困惑。下面提出的一些概念可能会对理解此概念有所帮助。

第23章的全称类型可从两种方式来看待。一个逻辑直觉就是类型 $\forall X.T$ 的一个元素一个具有对所有 S 都有类型 $[X \mapsto S]T$ 的值。这种直觉对应于一个类型抹除的观点:例如,多态恒等函数 $\lambda X.\lambda x{:}X.x$ 抹除为无类型恒等函数 $\lambda x.x$,可以将任意类型 S 的参数映射到相同的类型结果。相比之下,一个操作的直觉是 $\forall X.T$ 的元素为一个函数,它将类型 S 映射到一个特殊类型为 $[X \mapsto S]T$ 的项。该直觉对应的是第23章中系统 F 的定义,其中类型应用的归约被认为是实际的计算步骤。

类似地,也有两种不同的方式来看待存在类型,写成 $\{\exists X,T\}$。逻辑直觉是 $\{\exists X,T\}$ 的一个元素对某类型 S 来说,是类型 $[X \mapsto S]T$ 的值。另一方面,操作直觉是 $\{\exists X,T\}$ 的元素是序对,记为 $\{*S,t\}$,其中项 t 的类型为 $[X \mapsto S]T$。

在本章中,我们要强调存在类型的操作观点,因为它使得存在类型与编程语言中出现的模块与数据类型更加类似。具体的语法反映了这种类比:写为 $\{\exists X,T\}$,花括号强调了存在值是二元组形式,而不是标准的符号 $\exists X.T$。

为理解存在类型,还需要先知道两点:怎样建立或"引入"(是9.4节中的行话)能填满它们的元素,怎样在计算中使用(或消去)这些值。

存在类型值是用一个类型对组成的项表示的,记 $\{*S,t\}$[2]。一个实用的具体直觉是认为类型 $\{\exists X,T\}$ 的值 $\{*S,t\}$ 带一个(隐藏的)类型分量和一个项分量的包或模块[3]。类型 S 常被称为隐藏表示类型,或有时(为强调与逻辑的联系,参见9.4节)是包的可见类型。例如,包

---

[1] 本章学的系统大部分还是带存在类型的系统 F(参见图23.1)。例子也是使用记录(参见图11.7)和数字(参见图8.2)。相关的 OCmal 实现是 fullpoly。

[2] 我们用 * 来标注序对中的类型分量是为了避免与通常的项二元组(参见11.7节)相混淆。另一种写法是 pack X = S with t。

[3] 显然,可以想像将这些模块一般化为许多类型和/或项分量,但为保持概念的可读性还是对每个当中的一个进行一般化。对多类型分量,可以套入单个类型的存在量词,而对多项的分量可使用一个二元组或记录作为右端的分量:

$$\{*S_1, *S_2, t_1, t_2\} \overset{\text{def}}{=} \{*S_1, \{*S_2, \{t_1, t_2\}\}\}$$

p = {*Nat, {a = 5, f = λX:Nat.succ(x)}}有存在类型{∃X, {a:X, f:X→X}}。p 的类型分量是 Nat，值分量是一个记录，对某些 X(称为 Nat)该记录含类型 X 的字段 a 和类型为 X→X 的字段 f。

相同的包 p 也有类型{∃X, {a:X, f:X→Nat}}，因为它的右端的分量是一个记录，其中对某 X(称为 Nat)，字段 a 和 f 的类型分别为 X 和 X→Nat。该例子说明，通常，类型检查器不能自动决定一个给定的包是属于哪一个存在的类型：所以程序员必须指名是哪一个。最简单的方法就是给每个包增加一个注释，明确说明它需要的类型。因此，对存在类型的完整形式是如下方式：

```
p = {*Nat, {a=5, f=λx:Nat. succ(x)}} as {∃X, {a:X, f:X→X}};
▸ p : {∃X, {a:X,f:X→X}}
```

或者是(相同的包但不同的类型)：

```
p1 = {*Nat, {a=5, f=λx:Nat. succ(x)}} as {∃X, {a:X, f:X→Nat}};
▸ p1 : {∃X, {a:X,f:X→Nat}}
```

用 as 引入的类型注释与 11.4 节介绍的归属构造相似，因为该构造允许任何项都注释上它可以使用的类型。我们将归属说明作为包语法结构的一部分。存在性的类型规则如下所示：

$$\frac{\Gamma \vdash t_2 : [X \mapsto U]T_2}{\Gamma \vdash \{*U, t_2\} \text{ as } \{\exists X, T_2\} : \{\exists X, T_2\}} \qquad \text{(T-Pack)}$$

要注意的一点是，带不同隐藏表示类型的包可以采纳相同的存在类型。例如：

```
p2 = {*Nat, 0} as {∃X,X};
▸ p2 : {∃X, X}

p3 = {*Bool, true} as {∃X,X};
▸ p3 : {∃X, X}
```

或更有用的：

```
p4 = {*Nat, {a=0, f=λx:Nat. succ(x)}} as {∃X, {a:X, f:X→Nat}};
▸ p4 : {∃X, {a:X,f:X→Nat}}

p5 = {*Bool, {a=true, f=λx:Bool. 0}} as {∃X, {a:X, f:X→Nat}};
▸ p5 : {∃X, {a:X,f:X→Nat}}
```

24.1.1 练习[★]：以上是相同主题的三种变化形式：

```
p6 = {*Nat, {a=0, f=λx:Nat. succ(x)}} as {∃X, {a:X, f:X→X}};
▸ p6 : {∃X, {a:X,f:X→X}}

p7 = {*Nat, {a=0, f=λx:Nat. succ(x)}} as {∃X, {a:X, f:Nat→X}};
▸ p7 : {∃X, {a:X,f:Nat→X}}

p8 = {*Nat, {a=0, f=λx:Nat. succ(x)}} as {∃X, {a:Nat, f:Nat→Nat}};
▸ p8 : {∃X, {a:Nat,f:Nat→Nat}}
```

在什么情况下它们比 p4 和 p5 的作用要小些？

与模块的类比,还有助于直觉地理解存在量词消去结构。如果一个存在包对应一个模块,则包消去就像 open 或 import 指示:它允许模块中的成分在程序中的其他地方使用,但保持模块类型分量的相同抽象形式。这一点可以通过下面的模式匹配规则来做到:

$$\frac{\Gamma \vdash t_1 : \{\exists X, T_{12}\} \qquad \Gamma, X, x{:}T_{12} \vdash t_2 : T_2}{\Gamma \vdash \texttt{let}\ \{X,x\}{=}t_1\ \texttt{in}\ t_2 : T_2} \qquad \text{(T-Unpck)}$$

也就是说,如果 t_1 是产生存在包的表达式,则我们可以将其类型和项分量强加于模式变量 X 和 x,并用它们来计算 t_2(另一个关于存在消去的具体语法是 open t_1 as {X,x} in t_2)。

例如,类型{∃X,{a:X,f:X→Nat}}的包 p4(定义见前面所述)。消去表达为:

```
let {X,x}=p4 in (x.f x.a);

▸ 1 : Nat
```

解开包 p4,用它的字段(x.f 和 x.a)来计算一个数值。消去公式也能包含类型变量 X:

```
let {X,x}=p4 in (λy:X. x.f y) x.a;
▸ 1 : Nat
```

在包的类型检查过程中包的类型表示始终是抽象的,这一点说明对 x 的操作只能由它的"抽象类型"{a:X,f:X→Nat}来保证。尤其是,我们不能将 x.a 具体看成是一个数字:

```
let {X,x}=p4 in succ(x.a);
▸ Error: succ 的参数不是一个数字
```

这种限制比较合理,因为从上面看到有相同存在类型的包,如 p4 可能使用 Nat 或使用 Bool(或其他类型)作为替代类型。

还有一种更巧妙的使存在消去结构的类型检查失败的方法。在规则 T-Unpack 中,类型变量 X 会出现在计算 t_2 类型的上下文中,但不会出现在规则结论的上下文中。意思是说,结果类型 T_2 不会随便包含 X,因为 X 的任何自由发生都将超出结论的辖域:

```
let {X,x}=p in x.a;
▸ Error: Scoping error!
```

关于这一点将在 25.5 节中更详细地讨论。

存在性的计算规则也直接表示为:

$$\begin{array}{l}\texttt{let}\ \{X,x\}{=}(\{{*}T_{11}, v_{12}\}\ \texttt{as}\ T_1)\ \texttt{in}\ t_2 \\ \quad \longrightarrow [X \mapsto T_{11}][x \mapsto v_{12}]t_2\end{array} \qquad \text{(E-UnpackPack)}$$

如果 let 的第一个子表达式已经归约为一个具体的包,那么将把该包的分量用 t_2 中的变量 X 和 x 来代替。与模块比较起来,该规则可看成是一个链接步骤,在该步中,指向一个独立的编译过的模块,象征名字(X 和 x)会被模块的实际内容所取代。

因为类型变量 X 被该规则替换了,结果程序实际上取得了进入包内部的路径。这就是另一种我们见过多次的现象:随着计算的推进,表达式会变得"更可类型化"——尤其是一个不良类型的表达将归约为一个良类型的表达。

带存在类型的系统 F 扩展定义总结在图 24.1 中。

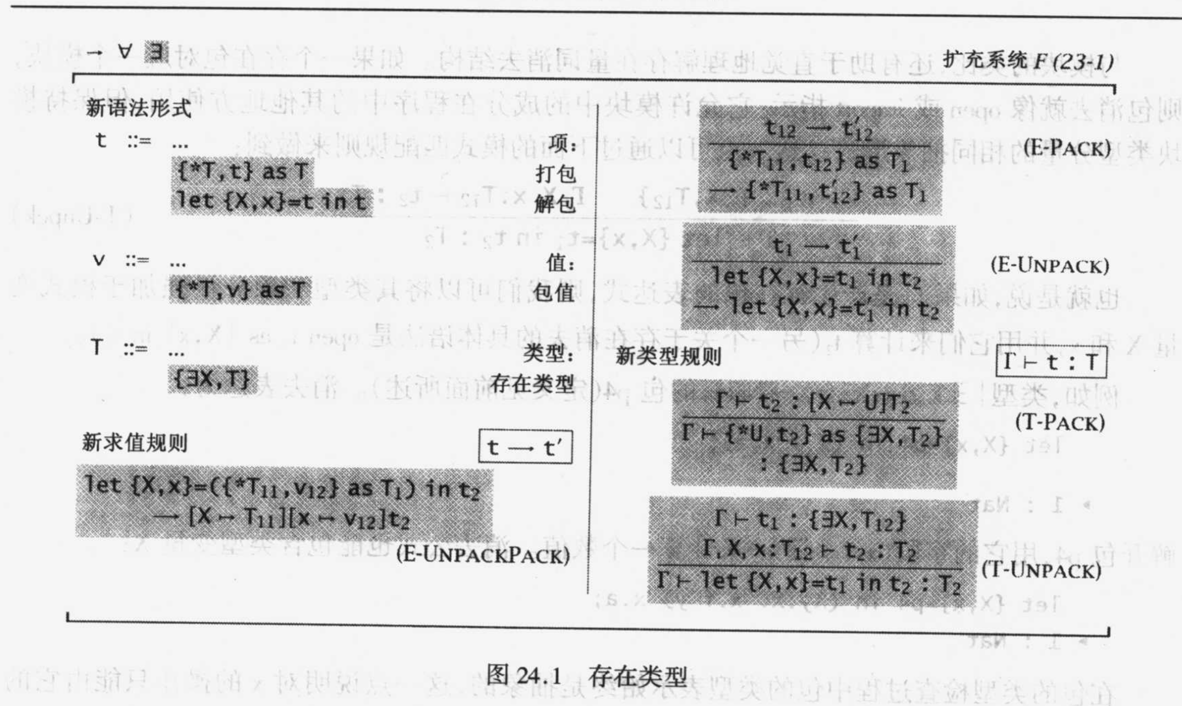


图 24.1 存在类型

24.2 带存在量词的数据抽象

在引言中(参见 1.2 节)说到过,类型系统在检测小范围的程序错误,如 2 + true 上超出了它的作用范围:但它们极大地为程序提供关键的支持。尤其是类型不仅可用来将类型抽象植入程序中,而且还可以用于程序员定义的抽象上,即不但防止程序影响机器,还保护程序的每个部分彼此不互相影响[①]。该部分考虑抽象的两种不同形式——典型的抽象数据类型和对象,将存在类型作为一般的结构来讨论。

不像第 18 章中遇到的对象编码,本节中的所有例子都是纯函数程序。这是一个纯说明性的选择:模块性和抽象的机制几乎与定义抽象状态的有无完全无关(练习 24.2.2 和练习 24.2.4 通过文中的某些纯函数的实例有状态形式来说明这一点)。这里提到纯函数实例的原因是:(1)该选择提示了我们的例子是处在一个更简单且花销更小的形式结构中;(2)有时运行纯函数的程序会使类型化问题更有趣(它们的解相应地也会变得更有启发性)。因为在命令性编程中,易变的变量提供了一个"旁路",使程序中相距较远的部分能直接通信。在纯函数程序中,程序不同部分之间交换的信息必须通过传递函数的参数和结果,在这里它对类型系统是"可见的"。在对象里尤其是这样的,但这会使我们推迟对一些重要特性(如子类型和继承)的处理,直到第 32 章中会提到可任意使用更强大的类型理论机制来解决这个问题。

抽象数据类型

传统的抽象数据类型(或 ADT)包括:(1)类型名称 A;(2) 具体表示类型 T;(3)类型 T 值的创建、查询和操作等有关的实现;(4)能封装类型表示和操作的抽象边界。在此边界范围中,类

① 为了公平,我们要说明的是类型并不是惟一地保护程序员定义的抽象方式。在无类型语言中,可通过函数闭包、对象或特殊目的的结构,如 MzScheme 的 unit(Flatt 和 Felleisen, 1998)来达到类似的效果。

型的元素可看到具体形式(带类型 T)。但在边界之外,只看得到抽象的形式,带类型 A。类型 A 的值可在数据结构的周围使用或在数据结构中存储等,但不能直接被检查或改变——对 A 的任何操作只能由 ADT 提供。

例如,这里有一个纯函数计数器 counter 的抽象数据类型的声明,用类似于 Ada(美国国防部,1980)或 Clu(Liskov 等,1981)语言写成的伪代码形式:

```
ADT counter =
  type Counter
  representation Nat
  signature
     new : Counter,
     get : Counter→Nat,
     inc : Counter→Counter;
  operations
     new = 1,
     get = λi:Nat. i,
     inc = λi:Nat. succ(i);

counter.get (counter.inc counter.new);
```

该抽象类型名为 Counter,它的具体形式是 Nat。操作的实现是具体地处理 Counter 的对象,如 Nat:new 是常量 1;inc 操作是后续函数;get 是恒等式。该 signature 部分说明了该操作应怎样外部使用,在它们的具体类型中,用 Counter 来替换 Nat 的某些实例。抽象边界范围从 ADT 关键词到终止分号;在程序的剩余部分(即最后一行),Counter 和 Nat 之间的联系被中断,所以惟一对常量 counter.new 处理的方式是把它当成一个 counter.get 或 counter.inc 的参数。

我们还能将该伪代码符号一个接一个改成存在演算符号。首先,要产生一个含 ADT 内部内容的存在包:

```
counterADT =
   {*Nat,
    {new = 1,
     get = λi:Nat. i,
     inc = λi:Nat. succ(i)}}
 as {∃Counter,
     {new: Counter,
      get: Counter→Nat,
      inc: Counter→Counter}};

▸ counterADT : {∃Counter,
                 {new:Counter,get:Counter→Nat,inc:Counter→Counter}}
```

然后,我们解包,引入类型变量 Counter 作为包的隐藏表示类型的占位符,并引入了可进行下列操作的项变量 counter:

```
let {Counter,counter} = counterADT in
counter.get (counter.inc counter.new);

▸ 2 : Nat
```

这种存在类型的表示方式比起语法上易懂的伪代码要难看懂些,但它们在结构上是相同的。

通常,打开存在包的 let 内包含了程序的整个剩余部分:

```
let {Counter,counter} = <counter package> in
<rest of program>
```

在剩余部分,类型名 Counter 可被当成是基类型而植入程序中。我们能定义对计数器操作的函数:

```
let {Counter,counter}=counterADT in
let add3 = λc:Counter. counter.inc (counter.inc (counter.inc c)) in
counter.get (add3 counter.new);
▸ 4 : Nat
```

我们还能定义新的包含计数器的抽象数据类型。例如,下面的程序定义了触发器的 ADT,其中用计数器(不是特别有效)来作为它的表示类型:

```
let {Counter,counter} = counterADT in

let {FlipFlop,flipflop} =
     {*Counter,
      {new    = counter.new,
       read   = λc:Counter. iseven (counter.get c),
       toggle = λc:Counter. counter.inc c,
       reset  = λc:Counter. counter.new}}
   as {∃FlipFlop,
       {new:    FlipFlop, read: FlipFlop→Bool,
        toggle: FlipFlop→FlipFlop, reset: FlipFlop→FlipFlop}} in

flipflop.read (flipflop.toggle (flipflop.toggle flipflop.new));
▸ false : Bool
```

这样,一个大的程序可以分解为一个长的 ADT 声明序列,每一个都使用它的前驱提供给它的类型和操作来实现自己,并将自己打包作为一个整洁的,定义正确的抽象给它的后继。

这里说的信息隐藏的一个关键是表示独立性。我们可以代换掉 Counter ADT 中可供选择的一个操作。例如,下面是一个含 Nat 记录型的内部表示,而不是单一的 Nat:

```
counterADT =
   {*{x:Nat},
    {new = {x=1},
     get = λi:{x:Nat}. i.x,
     inc = λi:{x:Nat}. {x=succ(i.x)}}}
  as {∃Counter,
      {new: Counter, get: Counter→Nat, inc: Counter→Counter}};

▸ counterADT : {∃Counter,
                {new:Counter,get:Counter→Nat,inc:Counter→Counter}}
```

可以完全放心的是,整个程序都能保持类型安全,因为程序的剩余部分除了使用 get 和 inc,其他情况是不能存取 Counter 的实例的。

实践表明,基于抽象数据类型的程序风格能对大系统的健壮性和可维护性产生巨大改善。关于这点有几点理由。首先,这种风格限制了程序改动的范围。正如我们上面见到的,可以用另一个 ADT 的实现来代换其中一个 ADT 的实现,主要是通过改变它的具体表示类型和操作的实现过程,而这样做不会影响程序的其他部分,因为存在包的类型规则已经保证了程序的剩余部分不会依赖于 ADT 的内部状态。其次,它提倡程序员尽量使 ADT 的型构(signature)越小越

好，以限制程序不同部分之间的依赖性。最后，也可能是最重要的一点，为使操作的型构清楚，它还要求程序员考虑设计抽象问题。

24.2.1　练习[推荐，★★★]：请按照上面例子的模式来定义一个数字的抽象数据类型栈(stack)，其中带操作 new，push，top(返回当前顶端的元素)，pop(返回一个去掉顶端元素的新栈)，以及 isempty。使用练习 23.4.3 介绍的 List 类型为暗含的表示方式。写一个简单的产生栈的主函数，实现的功能是将几个数字推入栈中，取出栈的顶元素。该练习最好是上机完成。使用 fullomega 检查器并拷贝文件 test.f 的内容(该文件中含 List 构造器和它的相关操作)到自己的输入文件中。

24.2.2　练习[推荐，★★]：建立一个可变计数器的 ADT，使用第 13 章定义的引用单元。将 new 从一个常量变成一个函数，要求使用 Unit 参数，返回一个 Counter，并将 inc 操作的结果类型改成 Unit 而不是 Counter(除了存在类型外，fullomega 检查器也提供引用)。

存在量词对象

在上面小节中看到的几个字"打包，然后解包"其实就是使用存在包的 ADT 风格程序所具的特点。一个包定义了一个抽象类型及其相关的操作，它一旦建立好后，马上就解开每个包，给抽象类型绑定一个类型变量并抽象地显露 ADT 的操作。在本节中，我们要说明一个对象形式的数据抽象的简单形式怎样能被视为基于存在的不同编程语言的惯用法。该对象模型会在第 32 章进一步说明。

这里再次使用简单的计数器作为执行的例子，如在上面的存在性 ADT 中及第 18 章和第 19 章介绍对象时用到它一样。而且还是用纯函数风格，使发送消息 inc 给计数器后不会改变该位置上的内部状态，而是会返回一个增加了内部状态后的新的计数器对象。

一个计数器对象包含了两个基本成分：一个数字(它的内部状态)，一对方法：get 和 inc，它们是用来操纵状态的。我们还需要确保状态能被查询或更新的惟一方式是使用这两个操作中的一个。为做到这一点，可以将状态和操作都包装在存在包中(抽象状态的类型)。例如，一个带值 5 的计数器对可写成：

```
c = {*Nat,
     {state = 5,
      methods = {get = λx:Nat. x,
                 inc = λx:Nat. succ(x)}}}
    as Counter;
```

其中：

```
Counter = {∃X, {state:X, methods: {get:X→Nat, inc:X→X}}};
```

为使用一个计数器对象的方法，我们打开存在包并将它的 methods 中的合适元素用于它的 state 字段上。例如，为得到 c 的当前值，可以写：

```
let {X,body} = c in body.methods.get(body.state);
▸ 5 : Nat
```

更一般地，还可定义一个简短的函数来"传送 get 消息"给计数器：

```
sendget = λc:Counter.
            let {X,body} = c in
            body.methods.get(body.state);
▸ sendget : Counter → Nat
```

要唤醒一个计数器对象的 inc 方法有点难度。如果仅是用和 get 一样的方法，则类型检查器会显示：

```
let {X,body} = c in  body.methods.inc(body.state);
▸ Error: Scoping error!
```

因为类型变量 X 在 let 的类型中是一个自由变量。的确，我们所写的也产生不了直接感受，因为 inc 方法的结果完全是一个内部状态，而不是一个对象。为了满足类型检查器及我们对 inc 应该做的事的理解，必须取出未使用过的内部状态，把它当成一个计数器的对象重新打包，使用相同的方法记录和相同的内部状态类型放进原始对象中：

```
c1 = let {X,body} = c in
       {*X,
        {state = body.methods.inc(body.state),
         methods = body.methods}}
     as Counter;
```

更一般地，"传送 inc 消息"给一个计数器，可写为：

```
sendinc = λc:Counter.
            let {X,body} = c in
              {*X,
               {state = body.methods.inc(body.state),
                methods = body.methods}}
              as Counter;
▸ sendinc : Counter → Counter
```

根据下面两个基本的操作，可以得到更复杂的计数器操作：

```
add3 = λc:Counter. sendinc (sendinc (sendinc c));
▸ add3 : Counter → Counter
```

24.2.3 练习[推荐，★★★]：将 Counter 对象作为 FlipFlop 对象内部的表示类型来实现 FlipFlop 对象，用上面介绍的 FlipFlop ADT 为模型。

24.2.4 练习[推荐，★★]：接着练习24.2.2，用 fullomega 检查器来实现 Counter 对象的状态变化。

对象与 ADT 的比较

前一节介绍的例子并不是面向对象程序的一个成熟模型。在第 18 章和第 19 章看到的许多特性包括子类型、类、继承和通过 self 与 super 的递归，在这里都没有了。我们将在第 32 章中再来补充这些特性，到那时，我们已经对类型系统增加了一些必要的改进。但在这些简单的对象与前面介绍的 ADT 之间，有必要做一些有趣的对比。

粗看一下，这两个程序段得出的是相反的结论：对有 ADT 的程序，在 ADT 建立后，包立即被打开；另一方面，若包是用来定义对象时，包与对象之间是紧密联系在一起的，直到需要将其中的一个方法用于访问内部状态时它们才必须被打开。

这种差异带来的后果是在两种风格下"计数器的抽象类型"指向不同的事物。在 ADT 风格的程序中，由客户代码（如 add3 函数）操纵的计数器值是暗含的表示类型（比如简单数字）元素。在对象风格的程序中，每个计数器就是一个包——不仅包含了数字，还包括了 get 和 inc 方法的实现代码。风格上的不同可反映出，在 ADT 风格中，类型 Counter 是由 let 结构引入的囿类型变量，而在对象风格中，Counter 代表整个存在类型：

```
{∃X, {state:X, methods: {get:X→Nat,inc:X→X}}}.
```

这样，在运行时，由计数器 ADT 产生的所有计数器的值只是相同的内部表示类型元素，而且只有一种计数器操作作用于该内部表示。比较之下，每个计数器对象带有自己的表示类型及在此类型上的方法集合。

对象与 ADT 之间的区别产生了许多有用的好处。其中明显的一个就是，因为每个对象都选择自己的表示并带有自己的操作，一个程序可自由地将相同对象类型的许多不同实现混在一起进行。这一点对出现的子类型和继承尤为方便：我们只要定义一个代表对象的一般化的类，在此基础上可以进行不同的修改，使每个应用都有自己稍微不同或完全不同的表示方式。因为这些改进了的类的实例，用的是大部分相同的类型，它们可以用大致相同的代码来运行，并可存放在一个列表中，等等。

例如，用户界面库中会定义一个总的 Window 类，其中有子类 TextWindow，ContainerWindow，ScrollableWindow，TitledWindow，DialogBox，等等。每个子类都包含了自己的特殊实例变量（例如，TextWindow 可以使用 String 实例变量来表示自己的当前内容，而 ContainerWindow 使用的是一系列 Window 对象），并提供一些特殊的操作，如 repaint 和 handleMouseEvent。另外，若定义 Window 为一个抽象数据类型（ADT），则得到的结构缺乏灵活性。Window 的具体表示类型还需要包括一个表示每种特殊窗口实例的变式类型（参见 11.10 节），该类型记载的是与该窗口类型相关的具体数据。有些操作，如 repaint 运行的是变式的一个 case 语句，实现的是合适的代码。如果窗口有许多具体的形式，则 Window 抽象数据类型的单一声明会变得越来越多而难以处理。

对象与 ADT 之间的另一主要的实用区别，还涉及了二进制操作的状态——这种操作是接纳相同抽象类型的两个或更多参量。为继续讨论区别的问题，我们要先明白两种二进制操作的区别：

- 一些二进制操作可以完全根据公认的两种抽象值上的操作来实现。例如，对计数器实现一个相等操作，我们只需要检查每个计数器的当前值（用 get 访问），比较取回的两个数字，也就是说，相等（equal）操作也能处于抽象边界之外，以保护计数器的具体表示。我们称这样的操作为弱二进制操作。
- 其他的二进制操作若没有对两种抽象值的表示方式具体的访问特权就不能实现。例如，我们正在实现一个表示数字集合的抽象。寻遍了一些算法书后，我们选择了一个具体的集合表示作为标签树，它服从一些特殊而复杂的不变式。对两个集合的有效 union 操作是需要能具体地看到它们，如树结构。然而，我们不愿在任何公共接口处将

具体表示显露给集合抽象。因此要重新修改一下 union 以使之能有权访问这两个参量，而普通的客户代码是无法访问它们的，也就是说 union 操作必须在抽象边界之内。我们称这样的操作为强二进制操作。

对我们考虑的两种抽象风格来说，弱二进制是一种容易处理的情况，因为我们是将它们放在抽象边界的里面还是外面都没有多大的区别。如果选择将其放在外面，它们简单地被定义为自立函数(取出合适的 ADT 对象或值)。若将它们放在 ADT 内部，实际上是一样的(它们会有一个具体的访问参量表示的方式，即使它们还用不上)。将二进制操作放在对象里只是稍微有点难度，因为对象的类型变成了递归的[①]：

```
EqCounter = {∃X, {state:X, methods: {get:X→Nat, inc:X→X,
                                      eq:X→EqCounter→Bool}}}
```

另一方面，强二进制操作不能在我们的模式中表示为对象的方法。我们只能用上面处理弱二进制的方式来表示它们的类型：

```
NatSet = {∃X, {state:X, methods: {empty:X, singleton:Nat→X,
                                   member:X→Nat→Bool,
                                   union:X→NatSet→X}}}
```

但没有令人满意的方法来实现这种类型的对象：对 union 操作的第二个参数，我们只知道它提供了 NatSet 操作，但这些并没有告诉我们该怎样找出它的元素以便于计算 union。

24.2.5 练习[★]：为什么不能使用下面的类型？

```
NatSet = {∃X, {state:X, methods: {empty:X, singleton:Nat→X,
                                   member:X→Nat→Bool,
                                   union:X→X→X}}}
```

总的说来，单独的 ADT 表示直接支持二进制操作，而对象的多种表示为了获得灵活性，不会使用二进制的方法。这些优点是互补的，任何一种风格都不能取代另一个。

关于本节的讨论还要注意一点，这些比较是针对本章中早先提出的简化对象“纯化论者”模式。现在主流面向对象的语言，如 C++ 和 Java 中，类的设计允许一些强二进制方法的形式，而且实际上是被描述为本章介绍的纯对象和纯 ADT 之间的折中。在这些语言中，对象的类型就是被实例化的类的名称，而且名字要与其他类的名字区别开，即使它们实现的操作是一样的(参见 19.3 节)。也就是说，这些语言中给定的对象类型是由相应的类声明给定的惟一实现。而且，语言中的子类只能将实例变元加入到那些从超类继承来的类型中。这些约束意味着，每个属于类型 C 的对象都拥有类 C(惟一)声明中定义的所有实例变元(也可能有更多)。如果对象方法是将另一个 C 作为参数并访问它的实例变元也是合理的，只要它使用的是 C 所定义的实例变量就可以了。这就允许强二元运算，例如被定义为方法的集合的并运算。这种混合对象模式已由 Pierce 和 Turner(1993)和 Katiyar 等(1994)形式化了，并由 Fisher 和 Mitcell (Fisher 和 Mithchell, 1996, 1998; Fisher, 1996b, a)更详细地加以了说明。

---

① 难度在于这种对象类型中的递归与继承有关。参见 Bruce 等(1996)。

24.3 存在量词编码

考虑到全称类型,23.4 节中的多态类型的序对编码方式使我们想到给存在量词类型进行类似地编码,尤其是存在类型的元素就是一个类型和一个值的序对:

```
{∃X,T}  ≝  ∀Y. (∀X. T→Y) → Y.
```

即,给定一个结果类型及其延续,一个存在包可看成是一个数据值,该值用来调用延续部分以产生一个结果值。延续部分取用两个参数(类型 X 和类型 T 的值),并将它们用于计算最终的结果。

有了存在类型的编码形式,打包和解包的代码就能得出了。为编码包:

```
{*S,t} as {∃X,T}
```

必须使用 S 和 t 来建立类型∀Y.(∀X.T→Y)→Y 的值。该类型是以全称量词开头的,类型体是一个箭头类型。这种类型的一个元素应以下面的两个抽象开始:

```
{*S,t} as {∃X,T}  ≝  λY. λf:(∀X.T→Y). ...
```

为完成这个任务,应该返回类型为 Y 的结果;显然,惟一的办法就是将 f 用到合适的参数上。首先,我们提供类型 S(很自然,这是此时惟一能使用的类型):

```
{*S,t} as {∃X,T}  ≝  λY. λf:(∀X.T→Y). f [S] ...
```

现在,类型应用 f[S]具有类型[X ↦ S](T→Y),也就是说,([X ↦ S]T)→Y。这样,我们可以将 t(根据规则 T-Pack,t 的类型为[X ↦ S]T)作为下一个参数:

```
{*S,t} as {∃X,T}  ≝  λY. λf:(∀X.T→Y). f [S] t
```

按要求,此时整个应用 f[S]t 的类型为 Y。

为给解包结构 let {X,x} = t_1 in t_2 编码,过程类似。首先,类型规则 T-Unpack 告诉我们,t_1 应有类型{∃X,T_{11}},t_2 应有类型 T_2(在限定 X 和 x:T_{11}的扩展上下文条件下),而 T_2 是我们所希望的整个 let…in…的表达式[①]。如在 23.4 节中的 Church 编码,这里的想法是将引入形式({*S,t})编码成一个"能自我消去"的活值。所以这里消去形式的编码应使用存在包 t_1 并将其应用为参数,足以产生期望得到类型为 T_2 的结果:

```
let {X,x}=t₁ in t₂  ≝  t₁ ...
```

t_1 的第一个参数应该为整个表达式期待的结果类型,即 T_2:

```
let {X,x}=t₁ in t₂  ≝  t₁ [T₂] ...
```

好了,应用 t_1[T_2]有类型(∀X.T_{11}→T_2)→T_2。即,如果能现在提供另一个类型(∀X.T_{11}→T_2)的参数,我们就能结束了。这种的参数可通过抽象关于变量 X 和 x 的语句体 t_2 来获得:

```
let {X,x}=t₁ in t₂  ≝  t₁ [T₂] (λX. λx:T₁₁. t₂).
```

到此时,编码完成。

① 严格地说,翻译要求实现类型信息的额外位数不出现在项语法中,所以我们所翻译的确实是类型推导,而不是项。类似的情况可参见 15.6 节中子类型的约束语义。

24.3.1 练习[推荐,★★ ↛]:请拿一张空白的纸,不看上面的编码,自己打草稿写出来。

24.3.2 练习[★★★]:该怎样证明存在量词的编码是正确的?

24.3.3 练习[★★★★]:我们能从另一个方向,根据存在类型来编码全称类型吗?

24.4 注释

ADT与存在类型之间的对应首先是由 Mitchell 和 Plotkin(1988)开始研究的,他们还留意了与对象之间的联系。Pierce 和 Turner(1994)详细阐述了该联系,有关细节和进一步的引用请参见第32章。对象与 ADT 之间的权衡关系也由 Reynolds(1975),Cook (1991),Bruce 等(1996)及其他许多学者研究过。尤其 Bruce 等(1996)对使用二进制方法进行了深入的讨论。

我们已经知道存在类型是怎样给自然类型理论提供一个简单的抽象数据类型模式。为更适应语言,如 ML 中出现的(功能更强大的)模式系统,大量的更复杂的机制已经在研究。关于该领域,最好先从以下学者的著作开始:Cardelli 和 Leroy(1990),Leroy(1994),Harper 和 Lillibridge(1994),Lillibridge(1997),Harper 和 Stone(2000),及 Crary 等(2002)。

类型结构是一个用来限制抽象程度的语法规则。

——John Reynolds(1983)

第 25 章　系统 F 的 ML 实现

现在我们来扩充第 10 章的 $\lambda_{\to}$ 实现，将第 23 章和第 24 章的全称和存在类型包括进来。因为定义该系统的规则是语法制导的（像 $\lambda_{\to}$ 本身，但不像带子类型或相等递归类型的演算），它的 OCaml 实现非常直接。实现 $\lambda_{\to}$ 最有趣的扩展是包含变量绑定（在量词中）的类型表示。为此，我们用到了第 6 章的 de Bruijn 索引。

25.1　类型的无名表示

刚开始，我们用类型变量及全称和存在量词来扩充类型的语法：

```
type ty =
    TyVar of int * int
  | TyArr of ty * ty
  | TyAll of string * ty
  | TySome of string * ty
```

这里的表示习惯与 7.1 节中项的表示是一样的。类型变量包括两个整型：第一个说明了到变量边界的距离，而第二个，像一致性检查样，说明了期望的上下文的总长度。量词是用囿变量的字符串名字来注释的，比如暗示是一个打印函数。

下一步是扩充上下文，让它带除了项变量外还带约束的类型变量，为做到这一点，要增加一个新的绑定（binding）类型结构：

```
type binding =
    NameBind
  | VarBind of ty
  | TyVarBind
```

在早期的实现中，NameBind 绑定器只是被解析函数和打印函数所使用。VarBind 构造器和以前一样带一个类型。新 TyVarBind 构造器不带额外的数据值，因为（不同于项变量）类型变量在该系统中不是用任何附加的假设来注释的。在含囿量词（参见第 26 章）或高级分类（参见第 29 章）的系统中，我们将为每个 TyVarBind 添上合适的注释。

25.2　类型移位和代换

因为类型含有变量，我们需要定义函数来进行类型的移位和代换。

25.2.1　练习[★]：参考定义（6.2.1）中的项移位函数，写一个移位类型变量的类似函数的数学定义。

在 7.2 节中，将项的移位和代换看成是两个独立的函数，还说明了从本书的网站上确实用一个可行的“映射”函数就能完成这两个任务的情况。同样类似的映射函数也能用来解决定义类型移位和代换。让我们现在就来看看这个映射函数。

基本的一点是移位和代换对所有的结构(除了变量)都有相同的处理方式。如果我们提取出它们对变量的处理,则它们就完全相同了。例如,这里有一个对类型的移位函数,我们将练习 25.2.1 的解转为 OCaml 形式:

```
let typeShiftAbove d c tyT =
  let rec walk c tyT = match tyT with
      TyVar(x,n) → if x>=c then TyVar(x+d,n+d) else TyVar(x,n+d)
    | TyArr(tyT1,tyT2) → TyArr(walk c tyT1,walk c tyT2)
    | TyAll(tyX,tyT2) → TyAll(tyX,walk (c+1) tyT2)
    | TySome(tyX,tyT2) → TySome(tyX,walk (c+1) tyT2)
  in walk c tyT
```

该函数的参数包含一个量值 d,通过它,自由变量可被移位,一个截参数 c,所有低于它的值都不能移位(这样是为避免变量囿同时被类型内的量词移位),还有一个待移位的类型 tyT。

现在,如果我们从 typeShiftAbove 中抽象出 TyVar 语句到新的参数 onvar 下,并删除在 TyVar 中惟一提到的参数 d,就获得了一个一般的映射函数:

```
let tymap onvar c tyT =
  let rec walk c tyT = match tyT with
    TyArr(tyT1,tyT2) → TyArr(walk c tyT1,walk c tyT2)
  | TyVar(x,n) → onvar c x n
  | TyAll(tyX,tyT2) → TyAll(tyX,walk (c+1) tyT2)
  | TySome(tyX,tyT2) → TySome(tyX,walk (c+1) tyT2)
  in walk c tyT
```

从这里可以通过以参数的形式代入 TyVar 语句(作为具有抽象的 c,x 和 n 的函数)来恢复移位函数:

```
let typeShiftAbove d c tyT =
  tymap
    (fun c x n → if x>=c then TyVar(x+d,n+d) else TyVar(x,n+d))
    c tyT
```

当内部的截参数为 0 时,也很容易定义 typeShiftAbove 的特殊形式:

```
let typeShift d tyT = typeShiftAbove d 0 tyT
```

我们还能实例化 tymap,实现在类型 tyT 中用类型 tyS 代换类型变量值 j 的操作:

```
let typeSubst tyS j tyT =
  tymap
    (fun j x n → if x=j then (typeShift j tyS) else (TyVar(x,n)))
    j tyT
```

在类型检查和求值过程中使用类型代换时,我们一直要代换 0 序(最外面的)变量,然后还要移位结果,以使变量最终消失。帮助函数为我们做到了这点:

```
let typeSubstTop tyS tyT =
  typeShift (-1) (typeSubst (typeShift 1 tyS) 0 tyT)
```

25.3 项

与项的处理方法类似。我们为扩充第 10 章的 term 数据类型,先得出消去全称类型和存在类型形式:

```
type term =
    TmVar of info * int * int
  | TmAbs of info * string * ty * term
  | TmApp of info * term * term
  | TmTAbs of info * string * term
  | TmTApp of info * term * ty
  | TmPack of info * ty * term * ty
  | TmUnpack of info * string * string * term * term
```

对项的移位和代换的定义与在第 10 章出现的类似。但考虑到要写出一个通用的映射函数，这里还是要先提一下，与我们在前面对类型的处理方法相同。映射函数形式如下：

```
let tmmap onvar ontype c t =
  let rec walk c t = match t with
    TmVar(fi,x,n) → onvar fi c x n
  | TmAbs(fi,x,tyT1,t2) → TmAbs(fi,x,ontype c tyT1,walk (c+1) t2)
  | TmApp(fi,t1,t2) → TmApp(fi,walk c t1,walk c t2)
  | TmTAbs(fi,tyX,t2) → TmTAbs(fi,tyX,walk (c+1) t2)
  | TmTApp(fi,t1,tyT2) → TmTApp(fi,walk c t1,ontype c tyT2)
  | TmPack(fi,tyT1,t2,tyT3) →
      TmPack(fi,ontype c tyT1,walk c t2,ontype c tyT3)
  | TmUnpack(fi,tyX,x,t1,t2) →
      TmUnpack(fi,tyX,x,walk c t1,walk (c+2) t2)
  in walk c t
```

注意，tmmap 带四个参数，比 tymap 多一个。要找出原因，注意到项可能包含两个不同的变量类型：项变量及项中存在于注释里的类型变量。所以在移位过程中，例如，有两种层次工作需要做：项变量和类型变量。当处理一个含类型注释的项结构时，ontype 参数会告诉项映射器该做些什么，如 TmAbs 情况所示。如果处理的是更大的程序，将会遇到更多这样的情况。

只要给 tmmap 合适的参数，就可定义项的移位：

```
let termShiftAbove d c t =
  tmmap
    (fun fi c x n → if x>=c then TmVar(fi,x+d,n+d)
                    else TmVar(fi,x,n+d))
    (typeShiftAbove d)
    c t

let termShift d t = termShiftAbove d 0 t
```

对项变量，要检查截参值，并构建一个新的变量，和在 typeShiftAbove 做法一样。对类型来说，我们将调用前面定义过的类型移位函数来处理。

项的代换函数也是类似的：

```
let termSubst j s t =
  tmmap
    (fun fi j x n → if x=j then termShift j s else TmVar(fi,x,n))
    (fun j tyT → tyT)
    j t
```

注意 termSubst 没有改变类型注释（类型不能包含项变量，所以项代换不会影响它们）。

我们还需要一个函数将一个类型代换为一个项，比如在类型应用的求值规则中：

$$(\lambda X.t_{12})\ [T_2] \longrightarrow [X \mapsto T_2]t_{12} \qquad \text{(E-TappTabs)}$$

还可以用项映射来定义它：

```
let rec tytermSubst tyS j t =
  tmmap (fun fi c x n → TmVar(fi,x,n))
        (fun j tyT → typeSubst tyS j tyT) j t
```

这时，传给 tmmap 用来处理项变量的函数是相同的(它只是重构建了以前的项变量)；当遇到了类型注释时，我们要对它实现类型代换。

最后，在处理类型时，我们用 eval 和 typeof 定义一个包含基本代换函数的更方便的函数：

```
let termSubstTop s t =
  termShift (-1) (termSubst 0 (termShift 1 s) t)
let tytermSubstTop tyS t =
  termShift (-1) (tytermSubst (typeShift 1 tyS) 0 t)
```

25.4 求值

下面对 eval 函数的扩展是对图 23.1 和图 24.1 中介绍的求值规则的直接转述。该工作是由前面定义的代换函数来完成的。

```
let rec eval1 ctx t = match t with
    ...
  | TmTApp(fi,TmTAbs(_,x,t11),tyT2) →
      tytermSubstTop tyT2 t11
  | TmTApp(fi,t1,tyT2) →
      let t1' = eval1 ctx t1 in
      TmTApp(fi, t1', tyT2)
  | TmUnpack(fi,_,_,TmPack(_,tyT11,v12,_),t2) when isval ctx v12 →
      tytermSubstTop tyT11 (termSubstTop (termShift 1 v12) t2)
  | TmUnpack(fi,tyX,x,t1,t2) →
      let t1' = eval1 ctx t1 in
      TmUnpack(fi,tyX,x,t1',t2)
  | TmPack(fi,tyT1,t2,tyT3) →
      let t2' = eval1 ctx t2 in
      TmPack(fi,tyT1,t2',tyT3)
    ...
```

25.4.1 练习[★]：为什么在第一个 TmUnpack 情况中需要 termShift?

25.5 类型化

typeof 函数的新语句也是直接从类型抽象和应用，以及存在量词的打包和解包类型规则变化而来的。下面我们将 typeof 的完整定义写出来，让新的 TmTAbs 和 TmTApp 语句能与通常的抽象和应用的旧语句比较一下。

```
let rec typeof ctx t =
  match t with
    TmVar(fi,i,_) → getTypeFromContext fi ctx i
  | TmAbs(fi,x,tyT1,t2) →
      let ctx' = addbinding ctx x (VarBind(tyT1)) in
      let tyT2 = typeof ctx' t2 in
      TyArr(tyT1, typeShift (-1) tyT2)
  | TmApp(fi,t1,t2) →
      let tyT1 = typeof ctx t1 in
      let tyT2 = typeof ctx t2 in
      (match tyT1 with
          TyArr(tyT11,tyT12) →
            if (=) tyT2 tyT11 then tyT12
            else error fi "parameter type mismatch"
        | _ → error fi "arrow type expected")
  | TmTAbs(fi,tyX,t2) →
      let ctx = addbinding ctx tyX TyVarBind in
      let tyT2 = typeof ctx t2 in
      TyAll(tyX,tyT2)
  | TmTApp(fi,t1,tyT2) →
      let tyT1 = typeof ctx t1 in
      (match tyT1 with
           TyAll(_,tyT12) → typeSubstTop tyT2 tyT12
         | _ → error fi "universal type expected")
  | TmPack(fi,tyT1,t2,tyT) →
      (match tyT with
          TySome(tyY,tyT2) →
            let tyU = typeof ctx t2 in
            let tyU' = typeSubstTop tyT1 tyT2 in
            if (=) tyU tyU' then tyT
            else error fi "doesn't match declared type"
        | _ → error fi "existential type expected")
  | TmUnpack(fi,tyX,x,t1,t2) →
      let tyT1 = typeof ctx t1 in
      (match tyT1 with
          TySome(tyY,tyT11) →
            let ctx' = addbinding ctx tyX TyVarBind in
            let ctx" = addbinding ctx' x (VarBind tyT11) in
            let tyT2 = typeof ctx" t2 in
            typeShift (-2) tyT2
        | _ → error fi "existential type expected")
```

最有趣的新语句是实现 TmUnpack 的。它包含了以下几步:(1)我们要检查子表达式 t_1,要确保它有一个存在类型$\{\exists X.T_{11}\}$;(2)用一个类型变量绑定 X 和一个项变量绑定 $x:T_{11}$来扩充上下文 Γ,并要检查 t_2 是否有类型 T_2;(3)将 T_2 中的自由变量的索引移位,降低两个,使它更符合原来的 Γ;(4)将结果类型作为整个 let…in…表达式的类型。

显然,如果 X 随意发生在 T_2 中,第(3)步的移位将产生一个无意义的带负索引的自由变量;在此处的类型检查肯定会失败。我们可通过重新定义 typeShiftAbove 来保证当它快要产生一个带负索引的类型变量时,发出一个错误的信号而不是返回不合理的结果。

```
let typeShiftAbove d c tyT =
  tymap
    (fun c x n → if x>=c then
                    if x+d<0 then err "Scoping error!"
                    else TyVar(x+d,n+d)
                 else TyVar(x,n+d))
    c tyT
```

无论什么时候我们计算的存在消去表达式 let {X,x} = t_1 in t_2 中的 t_2 的类型含囿类型变量 X 时，这项检查都会产生一个越界错误：

```
  let {X,x}=({*Nat,0} as {∃X,X}) in x;
▸ Error: Scoping error!
```

第26章　囿　量　词[1]

在编程语言中,许多关注的焦点都集中在单独考虑特征之间相对简单的联系上。本章开始介绍囿量词,对它的研究是在多态和子类型组合后有所升温,它能充分提高系统的表达能力和元理论的复杂度。我们所要学习的演算称为 $F_{<:}$("F 子"),自从 20 世纪 80 年代中期开始研究以来,它已经在编程语言研究中占据中心地位,尤其在面向对象领域处于基础研究的地位。

26.1　引言

将子类型与多态组合在一起的最简单方法就是把它们当成完全的正交特征,也就是说,认为一个新的系统为第 15 章到第 23 章介绍的系统合并。该系统在理论上是成立的,只要子类型与多态彼此独立,则它就是有用的。但是,一旦我们在同一个语言中都使用了它们,就需要找到一个更合适的方法将它们组合在一起。为了陈述,先让我们看一个十分简单的例子(在 26.3 节中还能看到别的例子,在第 27 章和第 32 章中看到一些更大、更实用的实例)。

设 f 是在具有数值字段 a 的记录上的恒等函数:

```
f = λx:{a:Nat}. x;

▸ f : {a:Nat} → {a:Nat}
```

若 ra 是带 a 字段的记录:

```
ra = {a=0};
```

然后就可将 f 应用于 ra,我们以前所见到的类型系统会产生一个相同类型的记录:

```
f ra;

▸ {a=0} : {a:Nat}
```

类似地,如果定义带两个字段 a 和 b 的记录:

```
rab = {a=0, b=true};
```

通过使用包含规则(T-Sub,参见图 15.1)将 f 应用于 rab,使 rab 的类型变成{a:Nat},以达到匹配 f 所期望的类型:

```
f rab;

▸ {a=0, b=true} : {a:Nat}
```

然而,得出的结果类型只含字段 a,如果检查到项(f rab).b 就会报错。换句话说,rab 经过恒等函数的处理后,得不到它的 b 字段!

---

① 本章大部分要学习的系统是纯 $F_{<:}$(参见图 26.1)。所用的例子也是使用记录和数字。相关的 OCaml 操作是 fullfsub 和 fullfomsub(fullfsub 对大多数例子来说已经足够了;而含带参数的类型简写,如 Pair 就需要用 fullfomsub)。

若用系统 F 的多态,可得到 f 的另一种表示:

```
fpoly = λX. λx:X. x;
▸ fpoly : ∀X. X → X
```

这时再将 fpoly 应用于 rab(及一个合适的类型参数)就能产生正确的结果:

```
fpoly [{a:Nat, b:Bool}] rab;
▸ {a=0, b=true} : {a:Nat, b:Bool}
```

但将 x 的类型变为一个变量过程中,忽略了本该要用到的信息。如假设我们要写另一种形式的 f,能返回它的原始参数和数字字段的后继函数:

```
f2 = λx:{a:Nat}. {orig=x, asucc=succ(x.a)};
▸ f2 : {a:Nat} → {orig:{a:Nat}, asucc:Nat}
```

然后,再用子类型,将 f2 用于 ra 和 rab,但后者会失去 b 字段。

```
f2 ra;
▸ {orig={a=0}, asucc=1} : {orig:{a:Nat}, asucc:Nat}
f2 rab;
▸ {orig={a=0,b=true}, asucc=1} : {orig:{a:Nat}, asucc:Nat}
```

但这次,多态也解决不了问题了。如果我们用变量 X 来代替 x 的类型,就会失去约束,即 x 必须为带 a 字段的记录,因为要用它来计算 asucc 字段结果。

```
f2poly = λX. λx:X. {orig=x, asucc=succ(x.a)};
▸ Error: Expected record type
```

在 f2 的类型中,我们想要表达的 f2 的操作是使用任何包含数字 a 字段的记录类型 R 的参数,返回含类型 R 和类型 Nat 的字段记录。我们可用子类型关系简要地说明:f2 用类型{a:Nat}的子类型 R 参数返回 一个含类型 R 的字段和类型 Nat 字段的记录。这种想法可通过在 f2ploy 的囿变量 X 引入子类型约束来形式化:

```
f2poly = λX<:{a:Nat}. λx:X. {orig=x, asucc=succ(x.a)};
▸ f2poly : ∀X<:{a:Nat}. X → {orig:X, asucc:Nat}
```

这就出现了所谓 $F_{<:}$ 系统的典型特点囿量词。

26.2 定义

通常,要将第 23 章系统 F 的类型和项与第 15 章的子类型关系组合起来,并精炼全称量词使其带子类型约束,这样就可得到 $F_{<:}$ 系统。囿量词也可以用我们将要在 26.5 节见到的类似方法进行定义。

在定义 $F_{<:}$ 的子类型关系上有两种合理的方式,它们与囿量词在规则的形式上不同(S-All):这两种方式,一个是更易处理但写法不灵活,称为核心(kernal)规则;另一个是表达能力更强但技术上尚存问题的全子类型规则。在下面的小节中,将详细讨论这两种形式,26.2 节的前几小节介绍核心变量,后几小节再介绍全变量。这样,为更精确地说明变量,我们所指的系统分别分成核心 $F_{<:}$ 和全 $F_{<:}$,而原来的不精确的说法 $F_{<:}$ 指的是两种系统的总称。

图 26.1 是核心 $F_{<:}$ 的完全定义,并且还强调了与前面系统的不同之处。

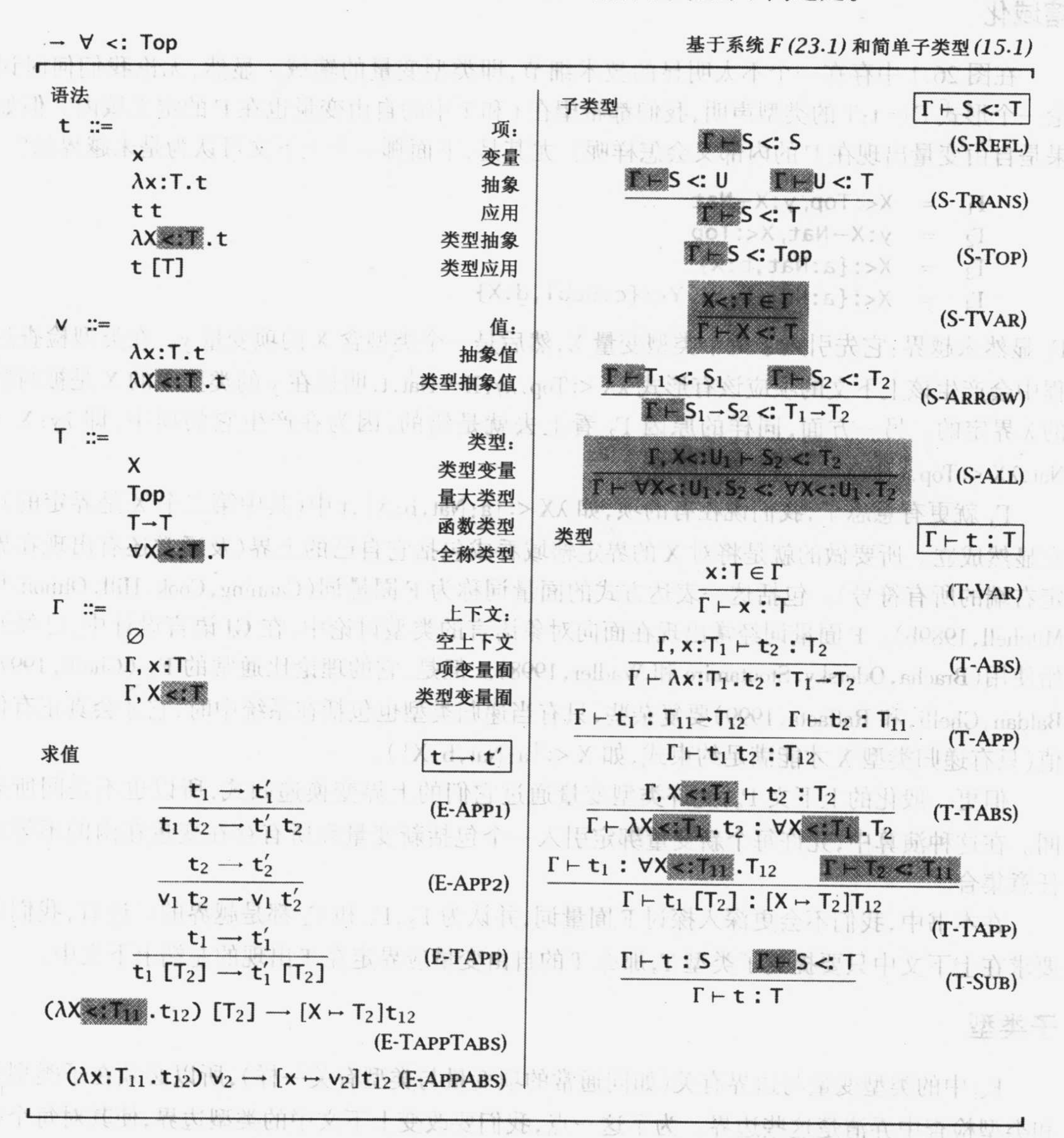


图 26.1 囿量词(核心 $F_{<:}$)

囿量词和非囿量词

从图中很明显可以看出 $F_{<:}$ 的语法提供了惟一的囿量词:一般地,纯系统 F 的未囿量词已经不存在了。因为我们并不需要它:一个囿量词边界为 Top,包括了 Top 的所有子类型,也就是所有的类型。所以我们可以采用如下的缩写来恢复非囿量词:

$$\forall X.T \stackrel{\text{def}}{=} \forall X{<:}Top.T$$

下面将经常使用该缩写。

辖域化

在图 26.1 中存在一个不太明显的技术细节，即类型变量的辖域。显然，无论我们何时讨论一个形式 $\Gamma \vdash t:T$ 的类型声明，我们都希望在 t 和 T 中的自由变量也在 Γ 的定义域内。但如果是自由变量出现在 Γ 的内部又会怎样呢？尤其是，下面哪一个上下文可认为是未越界的？

$$
\begin{aligned}
\Gamma_1 &= \texttt{X<:Top, y:X→Nat} \\
\Gamma_2 &= \texttt{y:X→Nat, X<:Top} \\
\Gamma_3 &= \texttt{X<:\{a:Nat,b:X\}} \\
\Gamma_4 &= \texttt{X<:\{a:Nat,b:Y\}, Y<:\{c:Bool,d:X\}}
\end{aligned}
$$

Γ_1 显然未越界：它先引入了一个类型变量 X，然后是一个类型含 X 的项变量 y。在类型检查过程中会产生该上下文的项应该有形式 $\lambda X <: Top.\lambda y:X \to Nat.t$，明显在 y 的类型中的 X 是被封装的 λ 界定的。另一方面，同样的原因 Γ_2 看上去就是错的，因为在产生它的项中，即 $\lambda y:X \to Nat.\lambda X <: Top.t$ 中 X 的范围不太明确。

Γ_3 就更有意思了，我们说在有的项，如 $\lambda X <: \{a:Nat,b:X\}.t$ 中(其中第二个 X 是界定的)，它显然成立。所要做的就是将对 X 的界定辖域看成包括它自己的上界(及通常还有出现在界定右端的所有符号)。包括这一表达方式的囿量词称为 F 囿量词(Canning, Cook, Hill, Olthoff 和 Mitchell, 1989b)。F 囿量词经常出现在面向对象语言的类型讨论中，在 GJ 语言设计中，已经开始使用(Bracha, Odersky, Stoutamire 和 Wadler, 1998)。但是，它的理论比通常的 $F_{<:}$ (Ghelli, 1997; Baldan, Ghelli, 和 Raffaetà, 1999)要复杂些，只有当递归类型也包括在系统中时，它才会真正有价值(只有递归类型 X 才能满足约束式，如 $X <: \{a:Nat,b:X\}$)。

但更一般化的上下文 Γ_4，允许类型变量通过它们的上界变换递归式，所以也不是闻所未闻。在这种演算中，允许每个新变量绑定引入一个包括新变量和所有存在变量在内的不等式任意集合。

在本书中，我们不会更深入探讨 F 囿量词，并认为 Γ_2, Γ_3 和 Γ_4 都是越界的。还有，我们还要求在上下文中只要提到了类型 T，那么 T 的自由变量应界定在 T 出现的左端上下文中。

子类型

$F_{<:}$ 中的类型变量与边界有关(如同通常的项变量与类型有关一样)，所以必须在子类型化和类型检查中弄清楚这些边界。为了这一点，我们要改变上下文中的类型边界，使其对每个类型变量都包含上界。在子类型化中，这些边界是用来判定“类型变量 X 是类型 T 的子类型，因为我们假设它是”。

$$\frac{X<:T \in \Gamma}{\Gamma \vdash X <: T} \tag{S-TVar}$$

增加该规则的意思是，子类型化有了三处关系，就是每个子类型化声明都有形式 $\Gamma \vdash S <: T$，读做“假设 Γ 成立，S 是 T 的子类型”。于是可在上下文加进所有其他的子类型化规则中来进行改进(参见图 26.1)。

除了变量的新规则，我们还必须增加一个比较量化类型的子类型化规则(S-All)。图 26.1 给出了一个更简单，称为核心规则的变量，它要求两个比较的量词必须相同。

$$\frac{\Gamma, X<:U_1 \vdash S_2 <: T_2}{\Gamma \vdash \forall X<:U_1.S_2 <: \forall X<:U_1.T_2} \quad \text{(S-All)}$$

术语"核心"(kernel)来自 Cardelli 和 Wegner 的原文(1985)，其中 $F_{<:}$ 的变式被称为 Kernel Fun。

26.2.1　练习[★ ↛]：设上下文为 $\Gamma = B <: Top, X <: B, Y <: X$，对 $B \to Y <: X \to B$ 画出子类型化推导树。

类型化

我们还要对通常的全称类型的类型规则进一步改进。这种扩展是直接的：在囿量词的引入规则中，对语句体进行类型检查时，把抽象中取出的边界放入上下文中：

$$\frac{\Gamma, X<:T \vdash t_2 : T_2}{\Gamma \vdash \lambda X<:T.t_2 : \forall X<:T.T_2} \quad \text{(T-TAbs)}$$

而在消去规则中，我们发现提供的类型参数的确满足边界要求：

$$\frac{\Gamma \vdash t_1 : \forall X<:T_{11}.T_{12} \qquad \Gamma \vdash T_2 <: T_{11}}{\Gamma \vdash t_1\ [T_2] : [X \mapsto T_2]T_{12}} \quad \text{(T-TApp)}$$

全 $F_{<:}$

在核心 $F_{<:}$ 中，只要两个量化类型的上界相同，它们就可以进行比较。如果我们把一个量词看成是一种箭头类型(它的元素是从类型到项的函数)，则核心规则对应的是一个箭头标准子类型规则的"协变"限制，它规定箭头类型的定义域不允许在子类型中变化：

$$\frac{S_2 <: T_2}{U \to S_2 <: U \to T_2}$$

该限制看上去对箭头和量词都不太合理。依次类推，我们应该改进一下核心 S-All 规则，允许囿量词的左端可以逆变子类型化，如图 26.2 中所示。

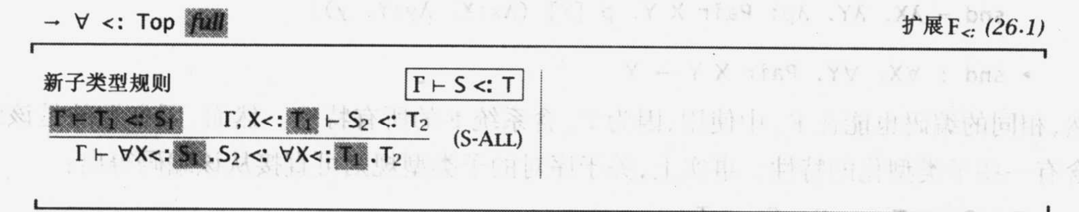


图 26.2　"全"囿量词

直观上，S-All 可以这样来理解。类型 $T = \forall X <: T_1.T_2$ 描述的是一类从类型变化到值的函数，每一个都是将 T_1 的子类型映射为 T_2 的实例。如果 T_1 是 S_1 的一个子类型，则 T 的定义域比 S 的定义域小，其中 $S = \forall X <: S_1.S_2$，因此 S 约束性更强，它描述了多态值的更小集合。而且，对每个在两种集合中都是函数可接受参数的类型 U(即可以满足更苛刻要求 $U <: T_1$)，如果

S_2 的 U 实例是 T_2 的 U 实例的子类型，则 S 是个“逐点变强”的约束，也描述了一个值的更小集合。

只带对量化类型的核心子类型化规则的系统称为核心 $F_{<:}$。带全量词子类型化的规则的系统就称为全 $F_{<:}$。而 $F_{<:}$ 本身是笼统地指这两种系统。

26.2.2 练习[★ ↛]：请给出一些类型的序对例子，使它们的关系满足全 $F_{<:}$ 的子类型关系，而不是核心 $F_{<:}$ 子类型。

26.2.3 练习[★★★★]：你能找到一些关于此特性的有用实例吗？

26.3 实例

本节展示了几个 $F_{<:}$ 的小的编程实例。它们都是用来说明该系统的特性，而不是说明它的具体应用；更大更复杂的例子可在后面的章节中(参见第 27 章和第 32 章)介绍。本章的所有例子都能在核心和全 $F_{<:}$ 中运行。

结果编码

在 23.4 节中，我们曾给出了在系统 F 中数序对的编码。该编码可被一般化为任意类型的序对：即下面类型的元素：

```
Pair T₁ T₂ = ∀X. (T₁→T₂→X) → X;
```

表示了 T_1 和 T_2 的序对。其构造器 pair 和析构器 fst 及 snd 的定义如下所示(关于 pair 定义的写法可帮助类型检查器以可读的方式打印出它的类型)：

```
pair = λX. λY. λx:X. λy:Y. (λR. λp:X→Y→R. p x y) as Pair X Y;
▸ pair : ∀X. ∀Y. X → Y → Pair X Y

fst = λX. λY. λp: Pair X Y. p [X] (λx:X. λy:Y. x);
▸ fst : ∀X. ∀Y. Pair X Y → X

snd = λX. λY. λp: Pair X Y. p [Y] (λx:X. λy:Y. y);
▸ snd : ∀X. ∀Y. Pair X Y → Y
```

显然，相同的编码也能在 $F_{<:}$ 中使用，因为 $F_{<:}$ 含系统 F 的所有特性。然而，更有趣的是该编码还含有一些子类型化的特性。事实上，关于序对的子类型规则可直接从该编码得出：

$$\frac{\Gamma \vdash S_1 <: T_1 \qquad \Gamma \vdash S_2 <: T_2}{\Gamma \vdash \text{Pair}\ S_1\ S_2 <: \text{Pair}\ T_1\ T_2}$$

26.3.1 练习[★ ↛]：证明上面的结论。

记录编码

有趣的是记录和记录型(包括它们的子类型化规则)能在纯 $F_{<:}$ 中编码。这里编码是由 Cardelli(1992)提出的。

我们先定义可变的元组。它们是“可变的”,因为它不同于普通的元组,它们可以在子类型化过程中向右扩充。

26.3.2 定义:对每个 $n \geq 0$ 和类型从 T_1 到 T_n,设:

$$\{T_i{}^{i \in 1..n}\} \stackrel{\text{def}}{=} \texttt{Pair } T_1 \ (\texttt{Pair } T_2 \ \ldots \ (\texttt{Pair } T_n \ \texttt{Top})\ldots).$$

特别地,$\{\} \stackrel{\text{def}}{=} \text{Top}$。类似地,对项从 t_1 到 t_n,设:

$$\{t_i{}^{i \in 1..n}\} \stackrel{\text{def}}{=} \texttt{pair } t_1 \ (\texttt{pair } t_2 \ \ldots \ (\texttt{pair } t_n \ \texttt{top})\ldots),$$

这里我们为了简短些,省略了 pair 的类型参数(这里的 top 就是 Top 中的任意元素,即任一个封闭的,良类型项)。投影 t.n(也省略了它的类型参数)是:

$$\texttt{fst } (\underbrace{\texttt{snd } (\texttt{snd} \ldots (\texttt{snd}}_{n-1\text{次}} \ \texttt{t})\ldots)$$

从该简写中,可立即得到下面关于子类型化和可变元组类型化的规则:

$$\frac{\Gamma \vdash {}^{i \in 1..n} S_i <: T_i}{\Gamma \vdash \{S_i{}^{i \in 1..n+k}\} <: \{T_i{}^{i \in 1..n}\}}$$

$$\frac{\Gamma \vdash {}^{i \in 1..n} t_i : T_i}{\Gamma \vdash \{t_i{}^{i \in 1..n}\} : \{T_i{}^{i \in 1..n}\}}$$

$$\frac{\Gamma \vdash t : \{T_i{}^{i \in 1..n}\}}{\Gamma \vdash t.i : T_i}$$

现在,设$\mathcal{L}$为标签的可数集合,其中的总体排序由双射(bijective)函数决定,该函数为 label-with-index:$\mathbb{N} \to \mathcal{L}$。我们按如下形式定义记录。

26.3.3 定义:设 L 为$\mathcal{L}$的有限集合,而 S_l 为每个 $l \in L$ 的类型。设 m 为 L 的元素的最大索引,且:

$$\hat{S}_i = \begin{cases} S_l & \text{如果 } \textit{label-with-index}(i) = l \in L \\ \text{Top} & \text{如果 } \textit{label-with-index}(i) \notin L. \end{cases}$$

如果记录型$\{l{:}S_l{}^{l \in L}\}$定义成可变元组$\{\hat{S}_i{}^{i \in 1..m}\}$,类似地,如果对每个 $l{:}L$,t_l 是一个项,则:

$$\hat{t}_i = \begin{cases} t_l & \text{如果 } \textit{label-with-index}(i) = l \in L \\ \text{top} & \text{如果 } \textit{label-with-index}(i) \notin L. \end{cases}$$

记录值$\{l = t_l{}^{l \in L}\}$是$\{\hat{t}_i{}^{i \in 1..m}\}$,投影 t.l 就是元组投影 t.i,其中 $\textit{label-with-index}(i) = l$。

该编码验证了类型化和子类型化规则的有效性(这些规则是图 15.2 和图 15.3 中的规则 S-RcdWidth, S-RcdDepth, S-RcdPerm, T-Rcd 和 T-Proj)。然而,这只是理论上有意义,站在实际的角度,它依赖的是所有标签范围内的全局顺序,这一点有严重的缺陷:因为在语言编辑中,数字分配给标签时不能一模块一模块地进行,而必须在链接时间内一次全部分配。

子类型化的 Church 编码

作为 $F_{<:}$ 表达的最后描述，让我们看看如果将囿量词增加到系统 F(参见 23.4 节)的 Church 数字编码中，将会发生什么事。那里的 Church 数字编码是：

```
CNat = ∀X. (X→X) → X → X;
```

用一个人性化的语言来解释该类型为："告诉我类型 T 的结果；现在给一个 T 上的函数及 T 的'基本元素'，然后将给你另一个 T 元素，该元素是在你所给的基本元素上重复实现你所给函数 n 次的结果"。

我们可以通过增加两个囿量词并加工一下 s 和 z 参数来一般化上面的编码：

```
SNat = ∀X<:Top. ∀S<:X. ∀Z<:X. (X→S) → Z → X;
```

直观地，该类型可读成："给我一个一般化的类型 X 和两个子类型 S 和 Z。现在给我一个函数将整个集合 X 映射到子集 S 和特殊集合 Z 的一个元素，这样就将返回一个 X 的元素，它是在基础元素上实现了函数 n 次后的结果。"

要弄清其中的道理，先看一下下面类型的轻微差异：

```
SZero = ∀X<:Top. ∀S<:X. ∀Z<:X. (X→S) → Z → Z;
```

尽管 SZero 与 SNat 在形式上几乎相同，但它透露出关于它所含元素的更重要内容，因为它能保证最终的结果将是 Z 的元素，而不光是 X 的元素。事实上，只有一种可能使它能发生，即它本身会产生一个参数 z。换句话说，值：

```
szero = λX. λS<:X. λZ<:X. λs:X→S. λz:Z. z;
▸ szero : SZero
```

是在类型 SZero 内的惟一值(也就是说 SZero 的元素的所有其他的元素的行为都和 szero 相同)。因为 SZero 是 SNat 的子类型，我们也可以写成 szero:SNat。

另外，相似的类型：

```
SPos = ∀X<:Top. ∀S<:X. ∀Z<:X. (X→S) → Z → S;
```

有更多的适用对象；例如：

```
sone   = λX. λS<:X. λZ<:X. λs:X→S. λz:Z. s z;
stwo   = λX. λS<:X. λZ<:X. λs:X→S. λz:Z. s (s z);
sthree = λX. λS<:X. λZ<:X. λs:X→S. λz:Z. s (s (s z));
```

等等。的确，SPos 适用于除 zsero 外的所有 SNat 的元素。

我们能类似地改进对 Church 数子定义的类型化操作。例如，类型系统可用检查出后继函数总是返回一个正数：

```
ssucc = λn:SNat.
          λX. λS<:X. λZ<:X. λs:X→S. λz:Z.
            s (n [X] [S] [Z] s z);
▸ ssucc : SNat → SPos
```

同样，通过改进它的参数类型，能写出函数 plus，让类型检查器提供给它类型 SPos→SPos→SPos。

```
spluspp = λn:SPos. λm:SPos.
                  λX. λS<:X. λZ<:X. λs:X→S. λz:Z.
                    n [X] [S] [S] s (m [X] [S] [Z] s z);
▸ spluspp : SPos → SPos → SPos
```

26.3.4 练习[★★]:请写出另一个 plus 的变式,除了类型注释外,该变式应与上面相同,其类型为 SZero→SZero→SZero。请写出一个类型为 SPos→SNat→SPos 的变式。

前面的例子与练习其实向我们提出了有趣的问题。显然,我们不愿意用不同的名称写出 plus 的不同形式,然后还要为该用哪种形式才是参数所要求的类型而发愁,所以我们需要使用单一规格的 plus,让它的类型含有所有可能的类型,比如:

```
plus :   SZero→SZero→SZero
       ∧ SNat→SPos→SPos
       ∧ SPos→SNat→SPos
       ∧ SNat→SNat→SNat
```

其中 $t:S \wedge T$,意思是"t 拥有 S 和 T 两种类型"。希望支持这种重载的想法使得我们去研究将囿量词与相交类型的组合系统。参见 Pierce(1997b)

26.3.5 练习[推荐,★★]:请按照 SNat 及其友元的模式,将 Church 布尔型(参见 23.4 节)中的 CBool 一般化为类型 SBool 和两个子类型 STrue 和 SFalse。请写出一个函数 notft,类型为 SFalse→STrue,及另一个类似的函数 nottf,类型为 STrue→SFalse。

26.3.6 练习[★ ↛]:在本章的介绍中发现子类型化和多态能以比在 $F_{<:}$ 中更直接,更不互交的方式组合在一起。我们从系统 F 开始(可能会加进记录等),并增加一个子类型关系(如在简单的带子类型化的 lambda 演算中),但量词要是非囿的。对子类型关系的惟一扩展应是对普通量词体的协变式子类型化规则:

$$\frac{S <: T}{\forall X.S <: \forall X.T}$$

请问:本章中哪一个例子能在这简单的系统中用公式表示?

26.4 安全

对 $F_{<:}$ 核心和全变量可直接建立类型保持特性。这里我们详细证明对核心 $F_{<:}$ 的情况;而对全 $F_{<:}$ 可以类推。然而,当我们在第 28 章中考虑子类型化和类型检查算法时,这两种变量要比本章给出的基本参数更加不同。我们将会发现在全系统中存在许多比核心系统复杂得多的难点,或者说,确实在全系统中缺少许多有用的特性(包括可决定的类型检查),而这些在核心系统中都具备。

我们将从类型和子类型关系的初步技术问题开始讨论。对它们的证明可按常规的推导进行。

26.4.1 引理[置换]:假设 Γ 是一个形式正确的上下文,而 Γ 是 Δ 的一种置换,即 Γ 有和 Δ 相同的绑定,Γ 中的顺序保持和 Δ 中类型变量相同的辖域,也就是说,如果在 Γ 中的一个绑定引入了一个在另一指向右端绑定的变量,则这些绑定会以同样的顺序出现在 Δ 中。

1. 如果 $\Gamma \vdash t:T$，那么 $\Delta \vdash t:T$。
2. 如果 $\Gamma \vdash S <: T$，那么 $\Delta \vdash S <: T$。

26.4.2 引理[弱化]：

1. 如果 $\Gamma \vdash t:T$ 和 $\Gamma, x:U$ 形式成立，则 $\Gamma, x:U \vdash t:T$。
2. 如果 $\Gamma \vdash t:T$ 且 $\Gamma, X <: U$ 成立，则 $\Gamma, X <: U \vdash t:T$。
3. 如果 $\Gamma \vdash S <: T$ 且 $\Gamma, x:U$ 成立，则 $\Gamma, x:U \vdash S <: T$。
4. 如果 $\Gamma \vdash S <: T$ 且 $\Gamma, X <: U$ 成立，则 $\Gamma, X <: U \vdash S <: T$。

26.4.3 练习[★]：对弱化引理的证明在哪点上要依赖于置换引理？

26.4.4 引理[在子类型推导中对项变量的强化]：如果 $\Gamma, x:T, \Delta \vdash S <: T$，则 $\Gamma, \Delta \vdash S <: T$。

证明：显然，类型假设在子类型推导中没起到任何作用。

通常，类型保持的证明都要依赖于与有类型化和子类型关系的代换的引理。

26.4.5 引理[狭窄化]：

1. 如果 $\Gamma, X <: Q, \Delta \vdash S <: T$ 且 $\Gamma \vdash P <: Q$，则 $\Gamma, X <: P, \Delta \vdash S <: T$。
2. 如果 $\Gamma, X <: Q, \Delta \vdash t:T$ 且 $\Gamma \vdash P <: Q$，则 $\Gamma, X <: P, \Delta \vdash t:T$。

这些特性通常都称为狭窄化，因为它们包括变量 X 的受限(狭窄)范围。

证明：留做练习[★]。

接下来，我们要得出关于代换和类型关系的引理。

26.4.6 引理[保持类型的代换]：如果 $\Gamma, x:Q, \Delta \vdash t:T$ 且 $\Gamma \vdash q:Q$，则 $\Gamma, \Delta \vdash [x \mapsto q]t:T$。

证明：使用上面介绍的特性对推导 $\Gamma, x:Q, \Delta \vdash t:T$ 进行归纳。

因为在归约的过程中需要对类型变量进行代换类型，我们还需要一个关系到类型代换和类型化的引理。该引理的证明(特别是 T-Sub 情况)要依赖一个与代换和子类型化有关的新引理。

26.4.7 定义：在上下文 Γ 所有绑定的右端将 S 代换为 X 而得到的上下文的形式记为 $[X \mapsto S]\Gamma$。

26.4.8 引理[类型代换保持子类型化]：如果 $\Gamma, X <: Q, \Delta \vdash S <: T$ 且 $\Gamma \vdash P <: Q$，那么 $\Gamma, [X \mapsto P]\Delta \vdash [X \mapsto P]S <: [X \mapsto P]T$。

注意，我们应该在 X 绑定的那部分中用 X 来代换，因为辖域约定要求 X 的绑定左端的类型不能包含 X。

证明：通过对 $\Gamma, X <: Q, \Delta \vdash S <: T$ 的推导归纳可知，惟一有趣的情况是最后两个：

情况 S-TVAR： $S = Y \quad Y <: T \in (\Gamma, X <: Q, \Delta)$

有两种子情况可以考虑。如果 $Y \neq X$，则结果立即由 S-TVar 得出。如果 $Y = X$，则我们有 $T = Q$ 且 $[X \mapsto P]S = Q$，所以结果可由 S-Refl 得出。

情况 S-ALL: $S = \forall Z<:U_1.S_2 \quad T = \forall Z<:U_1.T_2$
$\Gamma, X<:Q, \Delta, Z<:U_1 \vdash S_2 <: T_2$

通过归纳假设,$\Gamma,[X \mapsto P]\Delta, Z <: [X \mapsto P]U_1 \vdash [X \mapsto P]S_2 <: [X \mapsto P]T_2$。通过S-ALL,有:$\Gamma,[X \mapsto P]\Delta \vdash \forall Z <: [X \mapsto P]U_1.[X \mapsto P]S_2 <: \forall Z <: [X \mapsto P]U_1.[X \mapsto P]T_2$,就是说可得出 $\Gamma,[X \mapsto P]\Delta \vdash [X \mapsto P](\forall Z <: U_1.S_2) <: [X \mapsto P](\forall Z <: U_1.T_2)$。

下面是一个类似的类型代换和类型化相关的引理。

26.4.9 引理[类型代换保留类型]:如果 $\Gamma, X <: Q, \Delta \vdash t:T$ 且 $\Gamma \vdash P <: Q$,那么 $\Gamma,[X \mapsto P]\Delta \vdash [X \mapsto P]t:[X \mapsto P]T$。

证明:通过对 $\Gamma, X <: Q, \Delta \vdash t:T$ 的推导的归纳可知,这里我们只给出有意义的情况。

情况 T-TAPP: $t = t_1\ [T_2] \quad \Gamma, X<:Q, \Delta \vdash t_1 : \forall Z<:T_{11}.T_{12}$
$T = [Z \mapsto T_2]T_{12}$

通过归纳假设,$\Gamma,[X \mapsto P]\Delta \vdash [X \mapsto P]t_1:[X \mapsto P](\forall Z <: T_{11}.T_{12})$,也就是说,$\Gamma,[X \mapsto P]\Delta \vdash [X \mapsto P]t_1: \forall Z <: T_{11}.[X \mapsto P]T_{12}$。通过 T-TApp 可得,$\Gamma,[X \mapsto P]\Delta \vdash [X \mapsto P]t_1\ [[X \mapsto P]T_2]:[Z \mapsto [X \mapsto P]T_2]([X \mapsto P]T_{12})$,即,$\Gamma,[X \mapsto P]\Delta \vdash [X \mapsto P](t_1[T_2]):[X \mapsto P]([Z \mapsto T_2]T_{12})$。

情况 T-SUB: $\Gamma, X<:Q, \Delta \vdash t : S \Gamma, X<:Q, \Delta \vdash S <: T$

通过归纳假设,$\Gamma,[X \mapsto P]\Delta \vdash [X \mapsto P]t:[X \mapsto P]T$。通过代换下的子类型保持引理[参见引理(26.4.8)],可得 $\Gamma,[X \mapsto P]\Delta \vdash [X \mapsto P]S <: [X \mapsto P]T$,该结果可由 T-Sub 得出。

下面建立一些简单的子类型结构特性。

26.4.10 引理[子类型关系从右到左逆转]:

1. 如果 $\Gamma \vdash S <: X$,那么 S 是一个类型变量。
2. 如果 $\Gamma \vdash S <: T_1 \to T_2$,那么 S 或者是一个类型变量,或者具有形式 $S_1 \to S_2$,且满足 $\Gamma \vdash T_1 <: S_1$ 及 $\Gamma \vdash S_2 <: T_2$。
3. 如果 $\Gamma \vdash S <: \forall X <: U_1.T_2$,那么 S 或者是一个类型变量,或者具有形式 $\forall X <: U_1.S_2$ 满足 $\Gamma, X <: U_1 \vdash S_2 <: T_2$。

证明:第(1)点通过子类型推导很容易归纳出。惟一有趣的情况是规则 S-Trans,它是使用两个归纳假设,先在右端的前提上,然后在左端的前提上。只要将第(1)点放在传递性情况下,其他两点的判断类似。

26.4.11 练习[推荐,★★]:请说明一下以下“从左到右逆转”的特性:

1. 如果 $\Gamma \vdash S_1 \to S_2 <: T$,那么或者 T = Top,或者 $T = T_1 \to T_2$ 满足 $\Gamma \vdash T_1 <: S_1$ 及 $\Gamma \vdash S_2 <: T_2$。
2. 如果 $\Gamma \vdash \forall X <: U.S_2 <: T$,那么或者 T = Top,或者 $T = \forall X <: U.T_2$ 满足 $\Gamma, X <: U \vdash S_2 <: T_2$。
3. 如果 $\Gamma \vdash X <: T$,那么或者 T = Top,或者 T = X 或者 $\Gamma \vdash S <: T$ 满足 $X <: S \in \Gamma$。

4. 如果 $\Gamma \vdash \mathrm{Top} <: T$,那么 $T = \mathrm{Top}$。

可以依次使用引理(26.4.10)来建立一个类型关系的直接结构特性,这样可通过该特性来证明类型保留的关键情况。

26.4.12 引理:

1. 如果 $\Gamma \vdash \lambda x{:}S_1.s_2 : T$ 且 $\Gamma \vdash T <: U_1 \to U_2$,那么 $\Gamma \vdash U_1 <: S_1$ 且存在 S_2 满足 $\Gamma, x{:}S_1 \vdash s_2 : S_2$ 且 $\Gamma \vdash S_2 <: U_2$;
2. 如果 $\Gamma \vdash \lambda X <: S_1.s_2 : T$ 且 $\Gamma \vdash T <: \forall X <: U_1.U_2$,那么 $U_1 = S_1$ 且存在 S_2 满足 $\Gamma, X <: S_1 \vdash s_2 : S_2$ 且 $\Gamma, X <: S_1 \vdash S_2 <: U_2$。

证明:对归纳情况(规则 T-Sub)用引理(26.4.10)直接对类型推导进行归纳。

有了这些有利的条件,类型保持的证明是显而易见的。

26.4.13 定理[保持]:如果 $\Gamma \vdash t : T$ 且 $t \to t'$,那么 $\Gamma \vdash t' : T$。

证明:通过对 $\Gamma \vdash t : T$ 的推导的归纳假设,使用上面得出的结论可直接得出所有的情况。

情况 T-VAR, T-ABS, T-TABS: $t = x$, $t = \lambda x{:}T_1.t_2$, or $t = \lambda X<:U.t$

其实,上面的情况都不会发生,因为我们假设 $t \to t'$,且不存在关于变量,抽象或类型抽象的求值规则。

情况 T-APP: $t = t_1\ t_2$ $\Gamma \vdash t_1 : T_{11} \to T_{12}$ $T = T_{12}$ $\Gamma \vdash t_2 : T_{11}$

根据求值关系进行定义,会出现三种子情况:

子情况 1: $t_1 \to t_1'$ $t' = t_1'\ t_2$

则结果来自于归纳假设和 T-App。

子情况 2: t_1 is a value $t_2 \to t_2'$ $t' = t_1\ t_2'$

与上相同。

子情况 3: $t_1 = \lambda x{:}U_{11}.u_{12}$ $t' = [x \mapsto t_2]u_{12}$

可通过引理(26.4.12),$\Gamma, x{:}U_{11} \vdash u_{12} : U_{12}$ 对 U_{12} 有 $\Gamma \vdash T_{11} <: U_{11}$ 及 $\Gamma \vdash U_{12} <: T_{12}$。经过代换的类型保持[参见引理(26.4.6)],$\Gamma \vdash [x \mapsto t_2]u_{12} : U_{12}$,由此我们能通过 T-Sub 得到 $\Gamma \vdash [x \mapsto t_2]u_{12} : T_{12}$。

情况 T-TAPP: $t = t_1\ [T_2]$ $\Gamma \vdash t : \forall X<:T_{11}.T_{12}$
$T = [X \mapsto T_2]T_{12}$ $\Gamma \vdash T_2 <: T_{11}$

对演算关系进行定义,可得到两种子情况:

子情况 1: $t_1 \to t_1'$ $t' = t_1'\ [T_2]$

则结果来自于归纳假设和 T-TApp。

子情况 2: $t_1 = \lambda X<:U_{11}.u_{12}$ $t' = [X \mapsto T_2]u_{12}$

可通过引理(26.4.12),$U_{11}=T_{11}$且 $\Gamma,X<:U_{11}\vdash u_{12}:U_{12}$,满足 $\Gamma,X<:U_{11}\vdash U_{12}<:T_{12}$。通过代换时的类型保持[参见引理(26.4.6)],有 $\Gamma\vdash[X\mapsto T_2]u_{12}:[X\mapsto T_2]U_{12}$,由此我们通过引理(26.4.8)和 T-Sub 可得出 $\Gamma\vdash[X\mapsto T_2]u_{12}:[X\mapsto T_2]T_{12}$。

情况 T-SUB: $\Gamma\vdash t:S \quad \Gamma\vdash S<:T$

通过归纳假设有 $\Gamma\vdash t':S$,根据 T-Sub 结果得出。

该 $F_{<:}$ 的进展定理其实是对带子类型化的简单类型的 lambda 演算的一个直接扩充。通常,我们都是先记录一个典型形式的特性,该特性说明了关于箭头和量词类型封闭值的可能形式。

26.4.14 引理[典型形式]:

1. 如果 v 是类型 $T_1\to T_2$ 的一个封闭值,那么 v 有形式 $\lambda x:S_1.t_2$。
2. 如果 v 是类型 $\forall X<:T_1.T_2$ 的封闭值,那么 v 有形式 $\lambda X<:T_1.t_2$。

证明:两部分都可通过类型推导的归纳进行。我们只给第(2)部分参数。通过类型规则的检查,显然在推导中最后一条规则 $\vdash v:\forall X<:T_1.T_2$ 应为 T-TAbs 或 T-Sub。如果是 T-TAbs,那么规则的前提马上就能得出预期的结果。所以设想最后的规则是 T-Sub。从该规则的前提来看,我们有 $\vdash v:S$ 及 $S<:\forall X<:T_1.T_2$. 从逆转引理(26.4.10)来看,可知 S 有形式 $\forall X<:T_1\to S_2$。结果就是由归纳假设得出。

有了这个引理,对进展的证明就显而易见了。

26.4.15 定理[进展]:如果 t 是一个封闭类型正确的 $F_{<:}$ 项,那么或者 t 为一个值,或者存在 t′使 $t\to t'$。

证明:对类型推导进行归纳。变化的情况不会发生,因为 t 是封闭的。很快可以得出 lambda 抽象的两种情况,因为项和类型抽象都是值。对应用、类型应用和包含的情况都更有意义,但我们这里只讨论后两者(因为项应用类似于类型应用)。

情况 T-TAPP: $t=t_1\,[T_2] \quad \vdash t_1:\forall X<:T_{11}.T_{12}$
$\Gamma\vdash T_2<:T_{11} \quad T=[X\mapsto T_2]T_{12}$

通过归纳假设,或者 t_1 是一个值,或者它能形成求值的某一步。如果 t_1 能求值一步,则规则 E-TApp1 适用于 t。否则,如果 t_1 是一个值,那么典型形式引理(26.4.14)的第(2)部分说明 t_1 还有形式 $\lambda X<:T_{11}.t_{12}$。所有规则 E-TAppTAbs 适用于 t。

情况 T-SUB: $\Gamma\vdash t:S \quad \Gamma\vdash S<:T$

该结果可直接由归纳假设得出。

26.4.16 练习[★★★ ↛]:将本节的结论扩充到全 $F_{<:}$ 系统。

26.5 囿存在量词类型

我们可给存在类型(参见第 24 章)加上一个囿,如同我们为全称类型加上边界一样,这样就可得到如图 26.3 所示的囿存在量词。和囿全称词一样,子类型化规则 S-Some 会带来两种情况,其一是比较的两个量词的囿必须相同,另一种是它们可以不同。

26.5.1 练习[★]: 什么是 S-Some 的全变式?

26.5.2 练习[★]: 在带记录和存在类型(但没子类型)的纯系统 F 中,为使:

```
{*Nat, {a=5,b=7}} as T;
```

为良类型,该有多少种选择 T 的方式? 如果加进了子类型和囿存在量词,我们是不是有更多的选择方式?

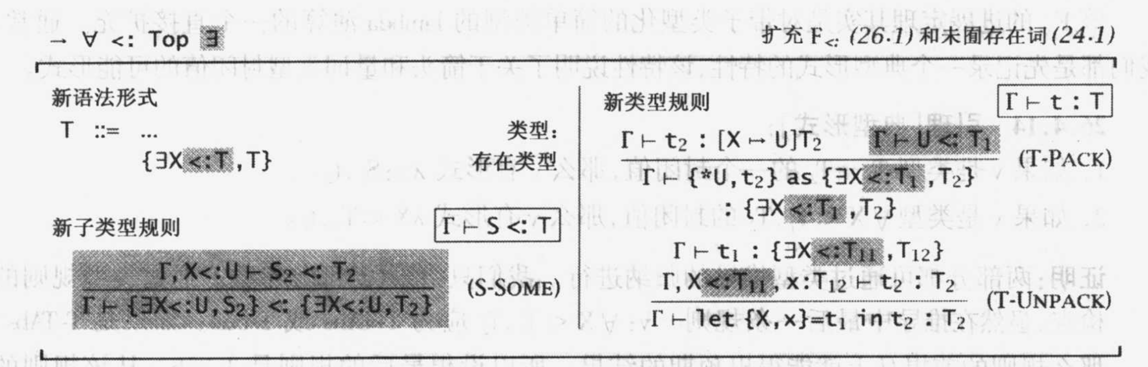


图 26.3 囿存在量词(核心变量)

在 24.2 节中见过一般的存在量词是如何用来实现抽象数据类型的。当我们给存在量词加上界时,能得到一个 ADT 层次的改进方式,Cardelli 和 Wegner(1985)给它取名为部分抽象类型。该想法的关键之处就是囿存在量词向外面的世界揭示了它表示的类型结构,但却隐藏了表示类型的真正身份。

例如,假设实现 24.2 节中的计数器的一个 ADT,但要在类型注释中增加界定 Counter <:Nat:

```
counterADT =
    {*Nat, {new = 1, get = λi:Nat. i, inc = λi:Nat. succ(i)}}
    as {∃Counter<:Nat,
         {new: Counter, get: Counter→Nat, inc: Counter→Counter}};

▸ counterADT : {∃Counter<:Nat,
                 {new:Counter,get:Counter→Nat,inc:Counter→Counter}}
```

我们能和前面一样使用计数器 ADT,将其类型和项成分绑定为变量 Counter 和 counter,然后用 counter 的字段来对计数器实现操作:

```
let {Counter,counter} = counterADT in
counter.get (counter.inc (counter.inc counter.new));

▸ 3 : Nat
```

而且,选择可以直接把 Counter 值当成数字:

```
let {Counter,counter} = counterADT in
succ (succ (counter.inc counter.new));

▸ 4 : Nat
```

另一方面,我们始终都不能将数字当成是 Counter:

```
let {Counter,counter} = counterADT in
counter.inc 3;
▸ Error: parameter type mismatch
```

用这种计数器抽象的形式是有效的,我们想通过显露它们的表示形式使外部世界使用时更简单些,但还是要保留计数器创建的控制方式。

26.5.3 练习[★★★]:假设我们要定义两种抽象数据类型,Counter 和 ResetCounter,使(1)这两种 ADT 都有操作 new,get 和 inc;(2)ResetCounter 另外提供 reset 操作,用来实现输入一个计数器返回一个新计数器集(含某不变值,如 1);(3)两种 ADT 的使用者都能用一个 ResetCounter 来代替一个 Counter(即有 ResetCounter <: Counter);(4)使用者对计数器和重置计数器内部是如何表示的一无所知。请问:如果是用囿存在量词包,这些能实现吗?

考虑到 24.2 节的存在量词,我们可以类似改进对象的编码方式。在 24.2 节中,已证实存在类型的包用来表示对象的内部状态类型,而这些对象是实例变量的记录。如果用囿存在量词代替非囿存在量词,对象的实例变量的一部分(但不是全部)名称和类型会向外部显露。例如,这里有个部分内部状态可见的计数器对象,对外可见其 x 字段,而限制了它的 private 字段可见性:

```
c = {*{x:Nat, private:Bool},
     {state = {x=5, private=false},
      methods = {get = λs:{x:Nat}. s.x,
                 inc = λs:{x:Nat,private:Bool}.
                          {x=succ(s.x), private=s.private}}}}
    as {∃X<:{x:Nat}, {state:X, methods: {get:X→Nat, inc:X→X}}};
▸ c : {∃X<:{x:Nat}, {state:X,methods:{get:X→Nat,inc:X→X}}}
```

与上面的部分抽象计数器 ADT 相同,这种计数器对象通过 get 可读取内部的值,或者可以直接到内部看到 x 字段的状态。

26.5.4 练习[★★]:请说明如何将按照24.3节的全称词写的存在词的编码扩充为按照囿全称词而写的囿存在词的编码形式。请检查编码中的子类型规则 S-Some 及囿全称词的子类型规则。

26.6 注释

CLU(Liskov 等,1997,1981;Schaffert,1978; Scheifler,1978)是最早带安全类型的囿量词的语言。CLU 的参数囿概念是基本的量化囿的量词形式(参见 26.2 节),该量词可用来一般化为多种类型参数。

这里提出的囿量词的思想是由 Cardelli 和 Wegner(1985)在语言 Fun 中介绍的。他们的"核心 Fun"与这里的 $F_{<:}$ 对应。基于 Cardelli 早期的非正式的想法及采用 Mitchell(1984b)研究的技术来进行公式化后,Fun 将 Girard-Reynolds 的多态(Girard,1972;Reynolds,1974)与 Cardelli 的子类型的一阶演算(1984)合为一体。最早的 Fun 经过 Bruce 和 Longo(1990)简化及稍微一般化,后来又由 Curien 和 Ghelli(1992)加工,产生了演算我们称之为全 $F_{<:}$。关于囿量词的最复杂的论文是 Cardelli,Martini,Mitchell 和 Scedrov(1994)的共同研究成果。

$F_{<:}$及其相关理论已由编程语言理论人员和设计者进行了广泛研究。Cardelli 和 Wegner 的论文给出了第一个使用囿量词的编程实例;还有更多研究出现在 Cardelli 对大量同类实例研究中(1988a)。Curien 和 Ghelli(1992,Ghelli,1990)阐述了许多 $F_{<:}$ 的语法特性。与系统十分接近的语义问题也由 Bruce 和 Longo (1990),Martini (1988),Breazu-Tannen,Coquand,Gunter 及 Scedrov (1991),Cardone (1989),Cardelli 和 Longo(1991),Cardelli,Martini,Mitchell 和 Scedrov(1994),Curien 和 Ghelli(1992,1991)和 Bruce 和 Mitchell(1992) 等研究过。$F_{<:}$ 已经过 Cardelli 和 Mitchell (1991),Bruce(Bruce,1991),Cardelli (1992)和 Canning,Cook,Hill,Olthoff 和 Mitchell(1989b)等扩充,可包含记录型及其他更丰富的概念。囿量词也在下列语言中都起着重要的作用,这些语言是:Cardelli 的编程语言 Quest(1991,Cardelli 和 Longo,1991),在 HP Labs (Canning,Cook,Hill 和 Olthoff,1989a; Canning,Cook,Hill,Olthoff 和 Canning,1990)中开发的 Abel 语言,以及更近一段时间设计的,如 GJ(Bracha,Odersky,Stoutamire 和 Wadler,1998),Pict(Pierce 和 Turner,2000)及 Funnel(Odersky,2000)语言,等等。

囿量词对代数类型的 Church 编码(参见 26.3 节)的作用是由 Ghelli(1990)和 Cardelli,Martini,Mitchell 和 Scedrov(1994)提出来的。

作为扩充带交叉类型的 $F_{<:}$(参见 15.7 节)是由 Pierce(1991b,1997b)研究的。Compagnoni 和 Pierce(1996)提出过高级系统的变量可用来对多继承的面向对象语言进行模块化;而 Compagnoni (1994)对其元理论的特性进行了分析。

第 27 章　实例分析：命令性对象，约式①

在第 18 章中，提出了简单类型演算中一系列有关记录、引用和子类型化的专业术语，构建了命令性面向对象编程风格的核心。在第 18 章的最后，我们阐述了如何提高对象运行时间效率，即把构建对象方法表的工作从方法执行时移到对象创建时完成。在本章中，将使用囿量词来进一步提高这个模型的效率。

18.12 节所阐述的关键思想是：当调用一个类时，就将传递 self 方法表的一个引用传递给这个类。该类则使用这个引用来定义自己的一些方法，随后，就可以通过这个引用来回查该类返回的完整方法。例如，如果 SetCounter 和 SetCounterRep 分别为公有接口和内部含 get，set 和 inc 方法的计数器对象的表示类型：

```
SetCounter = {get:Unit→Nat, set:Nat→Unit, inc:Unit→Unit};

CounterRep = {x: Ref Nat};
```

那么我们可以按照如下方法实现一个设置计数器类：

```
setCounterClass =
  λr:CounterRep. λself: Source SetCounter.
    {get = λ_:Unit. !(r.x),
     set = λi:Nat. r.x:=i,
     inc = λ_:Unit. (!self).set
                       (succ ((!self).get unit))};
▸ setCounterClass : CounterRep → (Source SetCounter) → SetCounter
```

使用 Source SetCounter 代替 Ref SetCounter 作为 self 的参数，这是因为，当我们在定义 setCounterClass 的子类时，子类中的 self 将是一个不同的类型。例如，InstrCounter 是计数器对象的接口，InstrCounterRep 描述了该类的内部数据类型：

```
InstrCounter = {get:Unit→Nat, set:Nat→Unit,
                inc:Unit→Unit, accesses:Unit→Nat}

InstrCounterRep = {x: Ref Nat, a: Ref Nat};
```

我们可以如下定义该类：

```
instrCounterClass =
  λr:InstrCounterRep. λself: Source InstrCounter.
  let super = setCounterClass r self in
  {get = super.get,
   set = λi:Nat. (r.a:=succ(!(r.a)); super.set i),
   inc = super.inc,
   accesses = λ_:Unit.
                !(r.a)};
```

---

① 本章中用到的实例均为含记录(参见图 15.3)和引用的 $F_{<:}$ 系统(参见图 13.1)中出现的项。相关联的 OCaml 实现是 fullfsubref。

```
▸ instrCounterClass : InstrCounterRep →
                      (Source InstrCounter) → InstrCounter
```

这里，参数 self 的类型是 Source InstrCounter，在构造 super 时为了把该 self 作为 setCounterClass 的 self 参数传递，需要把 Source InstrCounter 强制转换成 Source SetCounter。协变式 Source 允许这种强制性变换，而不变式 Ref 不允许。

然而，就像在 18.12 节最后所看到的那样，这个类模型的效率仍然不是最理想的。因为同一个方法表跟给定类所实例化的每一个对象都是密切相关的，我们只需要在创建类时只建立一次方法表，以后创建对象时重用它即可。这比较准确地反映了现实世界中面向对象语言的实现惯例，一个对象不携带任何方法而只是一个指向代表该类的数据结构的指针，而方法实际上就保存在这个代表该类的数据结构中[①]。

换句话说，以上定义的类参数的顺序(第一个是实例变量，然后是 self)颠倒了：在构造类表的时候需要用到 self 参数，而只有在方法真正被调用的时候才能用到记录类型的实例变量 r。如果把 self 作为第一个参数，就可以在传递 r 参数以前计算出方法表；我们可以部分地一次将类应用于它的 self 参数，一次完成该计算，并将计算出的方法表应用于多个实例变量记录，以复制多个方法表。具体地讲，可以按如下方式重写 SetCounterClass：

```
setCounterClass =
  λself: Source (CounterRep→SetCounter).
  λr:CounterRep.
    {get = λ_:Unit. !(r.x),
     set = λi:Nat. r.x:=i,
     inc = λ_:Unit. (!self r).set
                       (succ ((!self r).get unit))};

▸ setCounterClass : (Source (CounterRep→SetCounter)) →
                    CounterRep → SetCounter
```

这个版本跟前一个版本有三个重大的区别。第一，新版本把 self 放到了 r 的前面。第二，self 的类型由 SetCounter 变成了 CounterRep→SetCounter。这个改变是由第一条改动引起的，因为 self 的类型必须与该类返回的方法表的类型保持一致。第三，类中所有(!self)都改成(!self r)。这个改变是由第二条改动引起的。

新计数器的实例化函数定义如下：

```
newSetCounter =
  let m = ref (λr:CounterRep. error as SetCounter) in
  let m' = setCounterClass m in
  (m := m';
   λ_:Unit. let r = {x=ref 1} in m' r);

▸ newSetCounter : Unit → SetCounter
```

我们注意到，以上定义的前三行代码只在 newSetCounter 定义时进行了一次求值。在最后一行，求值以平凡抽象结束，可以反复使用该平凡抽象来创建对象。每次，系统都会为新的记录类型

① 事实上，现实世界中面向对象语言更加深入。类不再计算和存储完整的方法表，而只是存储新添加的或覆盖超类中原有方法的那些方法。所以，在我们看来，方法表从来就没有建立过。在调用方法时，我们从接受对象的实际类开始，简单地浏览一下类的分层结构，直到找到期望的方法定义为止。这种处于运行时间的查找对静态类型分析提出了一个棘手的问题，这里我们暂时不做处理。

变量 r 分配存储空间以及用 r 初始化方法列表 m'来创建新的对象。

遗憾的是,通过上述方式重新组织 setCounterClass,引入了状态类型 CounterRep 的一个逆变式,这给我们在定义 setCounterClass 的子类时带来了一定的困难。

```
instrCounterClass =
  λself: Source (InstrCounterRep→InstrCounter).
  let super = setCounterClass self in
  λr:InstrCounterRep.
  {get = (super r).get,
   set = λi:Nat. (r.a:=succ(!(r.a)); (super r).set i),
   inc = (super r).inc,
   accesses = λ_:Unit. !(r.a)};
▸ Error: parameter type mismatch
```

类型不匹配产生于 super 的定义,因为用传给子类的同一个 self 来实例化超类 setCounterClass。遗憾的是,当前的 self 是类型 InstrCounterRep→InstrCounter 的函数引用,不是 CounterRep→SetCounter 的子类型,因为箭头的左端是错误的方式。

我们再以另一个角度来表示这个困难,如再重写 setCounterClass,这次使用囿量词:

```
setCounterClass =
  λR<:CounterRep.
  λself: Source(R→SetCounter).
    λr: R.
      {get = λ_:Unit. !(r.x),
       set = λi:Nat.  r.x:=i,
       inc = λ_:Unit. (!self r).set (succ((!self r).get unit))};
▸ setCounterClass : ∀R<:CounterRep.
                        (Source (R→SetCounter)) → R → SetCounter
```

这个改变使得 setCounterClass 对其传递的参数 self 的要求不是很苛刻。人性化地讲,我们可以这样描述前一个版本的 setCounterClass,它告诉环境,"请给我传递一个能接受 CounterRep 作为参数的 self,我将用它来构造一个同样需要 CounterRep 参数的方法表"。新版本则说:"请告诉我正在构造的对象的表达类型 R,R 必须包含一个 x 字段,因为我需要用到它。然后给我一个能接受 R 作为参数 self 引用并能返回一个方法表,该表至少应包含 SetCounter 接口中的方法。我将构建并返回同种类型的其他表"。

在定义子类 instrCounterClass 时,我们能非常清晰地看到这个改变带来的效果。

```
instrCounterClass =
  λR<:InstrCounterRep.
  λself: Source(R→InstrCounter).
    λr: R.
      let super = setCounterClass [R] self in
        {get = (super r).get,
         set = λi:Nat. (r.a:=succ(!(r.a)); (super r).set i),
         inc = (super r).inc,
         accesses = λ_:Unit. !(r.a)};
▸ instrCounterClass : ∀R<:InstrCounterRep.
                          (Source (R→InstrCounter)) →
                          R → InstrCounter
```

在这个定义中,没有上一个定义出现的类型不匹配的情况。原因是在于这里受 super 界定的表达式中使用了包含规则,提升了 self 类型使之成为超类所接受的类型。在前面,我们试图去说明:

```
     Source(InstrCounterRep→InstrCounter)
 <:  Source(CounterRep→SetCounter),
```

该式为假。现在要说明的是:

```
Source(R→InstrCounter) <: Source(R→SetCounter),
```

该式为真。

新类的构造子与旧类的构造子非常相似。例如,这里是一个计数器(instrumented counter)子类的构造子:

```
newInstrCounter =
  let m = ref (λr:InstrCounterRep. error as InstrCounter) in
  let m' = instrCounterClass [InstrCounterRep] m in
  (m := m';
   λ_:Unit. let r = {x=ref 1, a=ref 0} in m' r);

▸ newInstrCounter : Unit → InstrCounter
```

惟一的区别就在于我们需要用实例记录变量的实际类型 InstrCounterRep 对 instrCounterClass 进行实例化。如前面所示,前三行代码只在生成 newInstrCounter 绑定时执行了一遍。

最后,这里给出一些测试实例来证明计数器正按我们预计的方式工作:

```
ic = newInstrCounter unit;
ic.inc unit;
ic.get unit;

▸ 2 : Nat

ic.accesses unit;

▸ 1 : Nat
```

27.1 练习[推荐,★★★]:本章中类的新编码依赖于 Source 类型构造子的协变性。如果语言中仅依靠囿量词和不变 Ref 构造子能取得同样的效率吗(比如,用同样的操作给出类的一个良类型的编码)?

第 28 章　囿量词的元理论[1]

本章将讨论 $F_{<:}$ 系统的子类型化和类型检查算法。我们将研究 $F_{<:}$ 系统的核心变式和全变式,两者表现有些不同。有些性质两者都有,但是全变式的部分性质更难证明一些,而有的性质在全 $F_{<:}$ 系统中完全消失——这是我们为 $F_{<:}$ 系统额外的可表达性所付出的代价。

首先在 28.1 节和 28.2 节中给出一个类型检查算法,这个算法在两个系统中都可以使用。然后考虑子类型检查,在 28.3 节中针对核心系统,在 28.4 节中针对全系统。在 28.5 节中,将继续讨论全 $F_{<:}$ 的子类型化问题,主要注重于对子类型不可判定性的讨论。28.6 节将告诉读者核心系统有合类型和交类型运算,而全系统则没有。28.7 节简要谈到了由囿存在量词所带来的一些问题,28.8 节对增加一个最小类型 Bot 所带来的影响进行了分析。

28.1　揭示

我们在 16.2 节中,讨论过带子类型化的简单类型 lambda 演算的类型检查算法,其核心思想是由项的每一个子项的最小类型计算出该项的最小类型。对于 $F_{<:}$ 系统,可以采取同样的基本思想,但是我们需要考虑到系统中由于类型变量的存在而产生的一个小问题。看下面的项:

```
f = λX<:Nat→Nat. λy:X. y 5;
▸ f : ∀X<:Nat→Nat. X → Nat
```

这个式子显然是良类型的,因为根据类型规则 T-Sub,应用 y 5 中的变量 y 的类型可提升为 Nat→Nat。但是 y 的最小类型是 X,而 X 不是箭头类型。为了找到整个应用的最小类型,需要找到 y 所具有的最小箭头类型,即最小箭头类型为类型变量 X 的超类型。并不奇怪,我们可以提升 y 的最小类型,直到它不再是一个类型变量时就找到了该超类型。

形式上,$\Gamma \vdash S \Uparrow T$(读做“在 Γ 下 S 揭示为 T”)表示 T 是 S 的最小不变超类型。通过不断提升变量类型产生揭示定义,参见图 28.1。

→ ∀ <: Top

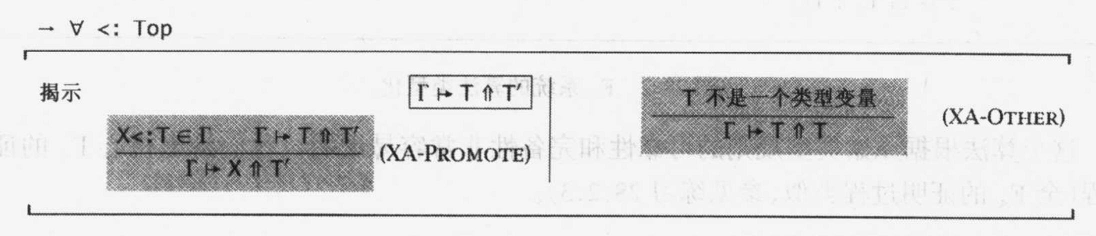


图 28.1　$F_{<:}$ 系统的揭示算法

很容易看出,这些规则定义了一个全函数。另外,揭示某个类型的结果通常是一个最小超

[1] 本章学的系统为纯 $F_{<:}$(参见图 26.1)。相应的实现为 purefsub;fullfsub 同时也包括了存在量词(参见图 24.1)及几个来自于第 11 章的扩展。

类型,它有一定的形状,不再是变量。比如,有 Γ = X <: Top, Y <: Nat→Nat, Z <: Y, W <: Z,我们可以得到:

$$\Gamma \vdash \text{Top} \Uparrow \text{Top} \qquad \Gamma \vdash \text{Y} \Uparrow \text{Nat}\to\text{Nat} \qquad \Gamma \vdash \text{W} \Uparrow \text{Nat}\to\text{Nat}$$
$$\Gamma \vdash \text{X} \Uparrow \text{Top} \qquad \Gamma \vdash \text{Z} \Uparrow \text{Nat}\to\text{Nat}$$

揭示的一些基本性质可以总结如下。

28.1.1 引理[揭示]: 假定 $\Gamma \vdash S \Uparrow T$,则:

1. $\Gamma \vdash S <: T$。
2. 如果 $\Gamma \vdash S <: U$ 并且 U 不是变量,则 $\Gamma \vdash T <: U$。

证明: 第(1)部分对推导 $\Gamma \vdash S \Uparrow T$ 的推导进行归纳,第(2)部分对推导 $\Gamma \vdash S <: U$ 的推导进行归纳。

28.2 最小化类型

计算最小类型的算法与带子类型化的简单类型的 lambda 演算建立在同一个基线上,只不过有一些曲折:当对一个应用进行类型检查时,我们需要计算左端的最小类型,然后揭示这个类型以获取箭头类型,如图 28.2 所示。如果左端的揭示不产生箭头类型,规则 TA-App 就不适用,而且应用也不是良类型的。类似地,我们通过揭示左端以得到量化类型对类型应用进行类型检查。

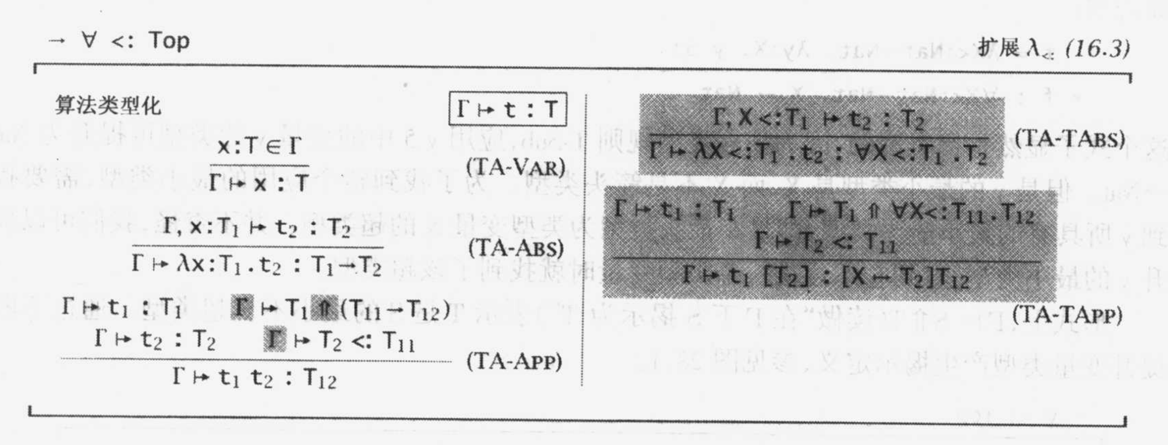


图 28.2 $F_{<:}$ 系统的算法类型化

这个算法根据原始类型规则的可靠性和完备性非常容易证明。我们给出核心 $F_{<:}$ 的证明过程(全 $F_{<:}$ 的证明过程类似,参见练习 28.2.3)。

28.2.1 定理 [最小类型化]:

1. 如果 $\Gamma \mapsto t:T$,则 $\Gamma \vdash t:T$。
2. 如果 $\Gamma \vdash t:T$,则 $\Gamma \mapsto t:M$ 且 $\Gamma \vdash M <: T$。

证明: 第(1)部分直接对算法推导进行归纳,根据引理(28.1.1)第(1)部分的应用情况。第(2)部分对 $\Gamma \vdash t:T$ 的一个推导做归纳,对推导中用到的最后一条规则进行情况分析。最有意思的情况是 T-App 和 T-TApp。

情况 T-VAR:　$t = x \quad x:T \in \Gamma$

由 TA-Var, $\Gamma \mapsto x:T$。由 S-Refl, $\Gamma \vdash T <: T$。

情况 T-ABS:　$t = \lambda x:T_1.t_2 \quad \Gamma, x:T_1 \vdash t_2 : T_2 \quad T = T_1 \to T_2$

根据归纳假设,对某些 M_2 满足 $\Gamma, x:T_1 \vdash M_2 <: T_2$,即 $\Gamma \vdash M_2 <: T_2$ 有 $\Gamma, x:T_1 \mapsto t_2 : M_2$,因为子类型化是不依赖于上下文中项变量绑定的[参见引理(26.4.4)]。由 TA-Abs,有 $\Gamma \mapsto t:T_1 \to M_2$。由 S-Refl 和 S-Arrow, $\Gamma \vdash T_1 \to M_2 <: T_1 \to T_2$。

情况 T-APP:　$t = t_1\ t_2 \quad \Gamma \vdash t_1 : T_{11} \to T_{12} \quad T = T_{12} \quad \Gamma \vdash t_2 : T_{11}$

由归纳假设,我们有 $\Gamma \mapsto t_1 : M_1$ 以及 $\Gamma \mapsto t_2 : M_2$,其中 $\Gamma \vdash M_1 <: T_{11} \to T_{12}$, $\Gamma \vdash M_2 <: T_{11}$。设 N_1 为 M_1 的最小不变超类型,即 $\Gamma \vdash M_1 \Uparrow N_1$。由引理(28.1.1)第(2)部分可知, $\Gamma \vdash N_1 <: T_{11} \to T_{12}$。我们知道 N_1 不是一个变量,由子类型关系的逆转引理(26.4.10)有 $N_1 = N_{11} \to N_{12}$,其中 $\Gamma \vdash T_{11} <: N_{11}$ 和 $\Gamma \vdash N_{12} <: T_{12}$。由传递性, $\Gamma \vdash M_2 <: N_{11}$。所以,规则 TA-App 得出 $\Gamma \mapsto t_1\ t_2 : N_{12}$,满足所有要求。

情况 T-TABS:　$t = \lambda X<:T_1.t_2 \quad \Gamma, X<:T_1 \vdash t_2 : T_2 \quad T = \forall X<:T_1.T_2$

由归纳假设,对某些 M_2 满足 $\Gamma, X <: T_1 \vdash M_2 <: T_2$,有 $\Gamma, X <: T_1 \mapsto t_2 : M_2$。由 TA-TAbs, $\Gamma \mapsto t: \forall X <: T_1.M_2$。由 S-All, $\Gamma \vdash \forall X <: T_1.M_2 <: \forall X <: T_1.T_2$。

情况 T-TAPP:　$t = t_1\ [T_2] \quad \Gamma \vdash t_1 : \forall X<:T_{11}.T_{12}$
$T = [X \mapsto T_2]T_{12} \quad \Gamma \vdash T_2 <: T_{11}$

由归纳假设,有 $\Gamma \mapsto t_1 : M_1$ 并且 $\Gamma \vdash M_1 <: \forall X <: T_{11}.T_{12}$。设 N_1 为 M_1 的最小不变超类型,即 $\Gamma \vdash M_1 \Uparrow N_1$。由引理(28.1.1)知, $\Gamma \vdash N_1 <: \forall X <: T_{11}.T_{12}$。我们知道 N_1 不是一个变量,由引理(26.4.10)子类型关系的逆转引理可知, $N_1 = \forall X <: T_{11}.N_{12}$,其中 $\Gamma, X <: T_{11} \vdash N_{12} <: T_{12}$。规则 TA-TApp 给出 $\Gamma \mapsto t_1[T_2]:[X \mapsto T_2]N_{12}$,根据代换子类型化保持引理(26.4.8)可知 $\Gamma \vdash [X \mapsto T_2]N_{12} <: [X \mapsto T_2]T_{12} = T$。

情况 T-SUB:　$\Gamma \vdash t : S \quad \Gamma \vdash S <: T$

由归纳假设, $\Gamma \mapsto t:M$ 其中 $\Gamma \vdash M <: S$。由传递性 $\Gamma \vdash M <: T$。

28.2.2　推论[类型化的可判定性]:给出子类型关系的一个判定过程,核心 $F_{<:}$ 系统的类型化关系就是可判定的。

证明:对任意 Γ 和 t,可以通过使用算法类型规则对 $\Gamma \mapsto t:T$ 进行证明,来检查是否存在某个 T 使得 $\Gamma \vdash t:T$。如果存在某个 T,那么由定理(28.2.1)的第(1)部分,T 在原始类型关系中也是 t 的类型。如果不存在,由定理(28.2.1)的第(2)部分,t 在原始类型关系中是无类型的。最后,注意到算法类型化规则对应于终止算法,因为它们是语法制导的(至多规则一次应用于一个项),当自底向上检查某项 t 时总能减少 t 的长度。

28.2.3　练习[★★]:对全 $F_{<:}$ 系统而言,上面的证明中哪些地方需要做修改?

28.3 核心 $F_{<:}$ 系统的子类型化

在 16.1 节中，我们注意到，带子类型的简单类型 lambda 演算中，声明性的子类型关系不是语法制导的，即它不能直接称为子类型化算法，有两点原因：(1)S-Refl 和 S-Trans 的结论与其他规则重叠(所以，自下而上地读这些规则，我们不知道应用哪一个)；(2)S-Trans 在前提中出现了一个元变量，而在结论中没有出现(这样初次使用的算法多少只能凭“猜测”)。解决这些问题非常简单，只要删除系统中这两条引起冲突的规则即可。进行删除以前，必须对这个系统进行修补，把这三条独立的子类型化规则合并成一条。

在这个方面，核心 $F_{<:}$ 系统的情况也基本一样。发生冲突的规则也是 S-Refl 和 S-Trans，可以删除这两条规则，并将剩下的规则做适当的修改，即将删除规则中基本有用的保留下来，这样可以得到一个新算法。

带子类型的简单类型 lambda 演算中，自反规则并没有什么实质性的用处，所以我们可以删掉它，而不需要改变那些可推导的子类型化声明[引理(16.1.2)，第(1)部分]。在 $F_{<:}$ 中，恰恰相反，形如 $\Gamma \vdash X <: X$ 的子类型化表达式却只能由自反性来证明。所以，当我们去掉自反规则时，必须增加一条受限制的自反公理，$\Gamma \vdash X <: X$，这条公理仅应用于变量。

$$\Gamma \vdash X <: X$$

类似地，为了消去 S-Trans，首先必须知道这条规则的哪些使用是最基本的。S-Trans 与 S-TVar 有一个有趣的联系，因为 S-TVar 允许类型变量的假设用于子类型语句的推导中。例如，$\Gamma = W <: Top, X <: W, Y <: X, Z <: Y$，如果删掉 S-Trans，就得不到 $\Gamma \vdash Z <: W$。看下面这个利用规则 S-Trans 的实例：

$$\dfrac{\dfrac{Z <: Y \in \Gamma}{\Gamma \vdash Z <: Y}\ (\text{S-TVAR}) \qquad \dfrac{\vdots}{\Gamma \vdash Y <: W}}{\Gamma \vdash Z <: W}\ (\text{S-TRANS})$$

在这个 S-Trans 实例中，左端的子推导是公理 S-Tvar 的实例。通常在这种情况下，S-Trans 不能被消去。

幸运的是，进行这种类型的推导是传递性的惟一基本用法。我们引入一个新的子类型规则：

$$\frac{X <: U \in \Gamma \qquad \Gamma \vdash U <: T}{\Gamma \vdash X <: T}$$

也能实现传递相同的功能。我们发现，由这条规则代替传递性和变量规则不会改变可推导子类型化的语句。

这些改变使我们提出了核心 $F_{<:}$ 系统的算法子类型关系，如图 28.3 所示。在算法类型化语句的十字转门符号上增加一个箭头以区别以往涉及这两者的类型语句。

事实上，新的规则 SA-Refl-TVar 和规则 SA-Trans-TVar 足以取代老的自反性和传递性规则，关于这一点以下引理可以说明：

28.3.1　引理[算法子类型关系的自反性]：对每一个 Γ 和 T，都有 $\Gamma \mapsto T <: T$ 成立。

证明：对 T 做归纳。

28.3.2　引理[算法子类型关系的传递性]：如果 $\Gamma \mapsto S <: Q$ 且 $\Gamma \mapsto Q <: T$，则 $\Gamma \mapsto S <: T$。

证明：对两个推导的总长度进行归纳。我们对给定的这两个推导中的最后规则进行情况分析。

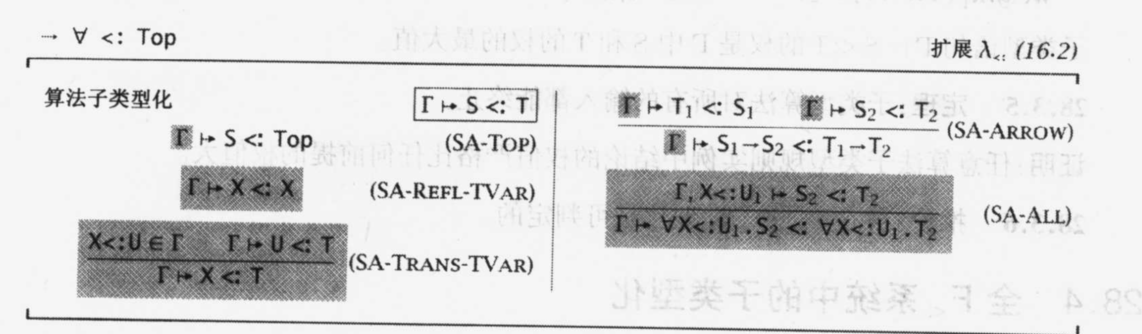


图 28.3　核心 $F_{<:}$ 系统算法的子类型化

如果右端推导 $\Gamma \mapsto Q <: T$ 是 SA-Top 的实例，则无须证明，结论成立，因为根据 SA-Top 有 $\Gamma \vdash S <: Top$。如果左端推导 $\Gamma \mapsto S <: Q$ 是 SA-Top 的实例，则 Q = Top，且由类型系统规则知道，右端推导 $\Gamma \mapsto S <: T$ 也必须是 SA-Top 的实例。

如果两边推导都是 SA-Refl-TVar 类型的实例，则也无须证明，因为另一个推导就是结论。

如果左端推导是 SA-Trans-TVar 类型，有 S = Y 且 $Y <: U \in \Gamma$，则可得出子推导 $\Gamma \mapsto U <: Q$，再由归纳假设，$\Gamma \mapsto U <: T$，根据 SA-Trans-TVar，得到 $\Gamma \mapsto Y <: T$。

如果左端推导结束时为 SA-Arrow 的实例，我们有 $S = S_1 \to S_2$ 和 $Q = Q_1 \to Q_2$，子推导为 $\Gamma \mapsto Q_1 <: S_1$ 和 $\Gamma \mapsto S_2 <: Q_2$。上面我们已经讨论过右端推导是 SA-Top 的情况，剩下惟一的可能性就是该推导也是以 SA-Arrow 结束的。因此，有 $T = T_1 \to T_2$ 以及 $\Gamma \mapsto T_1 <: Q_1$ 及 $\Gamma \mapsto Q_2 <: T_2$，两次使用归纳假设，得到 $\Gamma \mapsto T_1 <: S_1$ 和 $\Gamma \mapsto S_2 <: T_2$，再由 SA-Arrow，得到 $\Gamma \mapsto S_1 \to S_2 <: T_1 \to T_2$。

还有一种情况，左端推导以 SA-All 实例结束，证明过程与上面类似。有 $S = \forall X <: U_1 . S_2$ 及 $Q = \forall X <: U_1 . Q_2$，其中子推导 $\Gamma, X <: U_1 \mapsto S_2 <: Q_2$。同样，我们已经认为右端推导为 TSA-Top 的情况，所以必须以 SA-All 结束。所以 $T = \forall X <: U_1 . T_2$，其中子推导 $\Gamma, X <: U_1 \mapsto Q_2 <: T_2$。应用归纳假设得到 $\Gamma, X <: U_1 \mapsto S_2 <: T_2$。应用规则 SA-All 得到 $\Gamma \mapsto \forall X <: U_1 . S_2 <: \forall X <: U_1 . T_2$，由此得证。

28.3.3　定理：[算法子类型化的可靠性和完备性]：$\Gamma \vdash S <: T$ 当且仅当 $\Gamma \mapsto S <: T$。

证明：在两个方向上进行归纳。可靠性(⇐)常规可证。完备性(⇒)的证明依赖于引理(28.3.1)和引理(28.3.2)。

最后，我们需要检查子类型规则定义的算法是否完全，也就是说该算法是否对所有的输入

都能终止。可以这样来检查,为每一个子类型语句赋一个权值,然后检查一下有结论的那些算法规则的权值是不是严格比它们的前提的权值大。

28.3.4 定义: 在上下文 Γ 中,类型 T 的权写做 $weight_\Gamma(T)$,定义如下:

$$\begin{aligned} weight_\Gamma(X) &= weight_{\Gamma_1}(U) + 1 \quad \text{如果 } \Gamma = \Gamma_1, X<:U, \Gamma_2 \\ weight_\Gamma(Top) &= 1 \\ weight_\Gamma(T_1 \to T_2) &= weight_\Gamma(T_1) + weight_\Gamma(T_2) + 1 \\ weight_\Gamma(\forall X<:T_1.T_2) &= weight_{\Gamma, X<:T_1}(T_2) + 1 \end{aligned}$$

子类型语句 $\Gamma \vdash S <: T$ 的权是 Γ 中 S 和 T 的权的最大值。

28.3.5 定理: 子类型算法对所有的输入都能终止。

证明: 任意算法子类型规则实例中结论的权值严格比任何前提的权值大。

28.3.6 推论: 核心 $F_{<:}$ 中子类型化是可判定的。

28.4 全 $F_{<:}$ 系统中的子类型化

全 $F_{<:}$ 系统的子类型检查规则与核心 $F_{<:}$ 系统的规则基本一致,如图 28.4 所示。惟一的区别在于其用更加灵活的变式取代了 SA-All。对于核心 $F_{<:}$ 系统而言,原始子类型关系之上的算法关系可靠性和完备性与这些算法关系具有自反性和传递性直接对应。

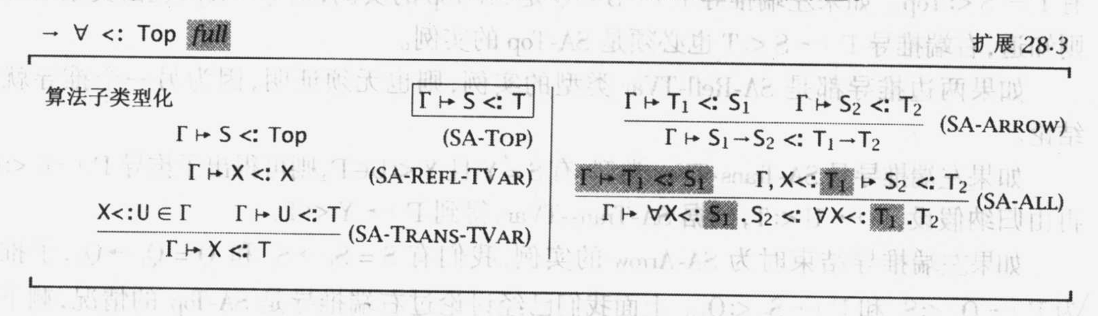


图 28.4 全 $F_{<:}$ 系统的算法的子类型化

对于自反性,结论与上面完全相同。而传递性的证明有一些麻烦。我们回忆一下上节中的核心 $F_{<:}$ 系统的传递性[引理(28.3.2)]的证明过程。其证明思想是给出以语句 $\Gamma \vdash S <: Q$ 和 $\Gamma \vdash Q <: T$ 结束的两个子类型推导,以及重新安排及设置它们的子推导构造出 $\Gamma \vdash S <: T$,而不使用传递性规则,并假设(作为一个归纳假设)对更小的推导做法也是相同的。如下例所示,假定我们有两个推导,都以新的 SA-All 规则结束:

$$\frac{\begin{array}{c}\vdots\\ \hline \Gamma \vdash Q_1 <: S_1\end{array} \quad \begin{array}{c}\vdots\\ \hline \Gamma, X<:Q_1 \vdash S_2 <: Q_2\end{array}}{\Gamma \vdash \forall X<:S_1.S_2 <: \forall X<:Q_1.Q_2} \qquad \frac{\begin{array}{c}\vdots\\ \hline \Gamma \vdash T_1 <: Q_1\end{array} \quad \begin{array}{c}\vdots\\ \hline \Gamma, X<:T_1 \vdash Q_2 <: T_2\end{array}}{\Gamma \vdash \forall X<:Q_1.Q_2 <: \forall X<:T_1.T_2}$$

为了跟以前的证明风格保持一致,我们将结合左右端的子推导,使用归纳假设,并最后使用 SA-All 规则来推导出结论 $\Gamma \vdash \forall X <: S_1.S_2 <: \forall X <: T_1.T_2$。左端的子推导没有问题,归纳假设告诉我们无传递性推导 $\Gamma \vdash T_1 <: S_1$。但是归纳假设对右边的子推导不适用,因为子推导所处的上下文是不同的,X 的上界在其中一个为 Q_1,而在另一个为 T_1。

幸运的是，我们知道如何使上下文保持一致。第 26 章中引理(26.4.5)狭窄化性质告诉我们，如果用界定的一个子类型来代替上下文中的界定，子类型声明将保持有效。似乎我们可以把 $\Gamma, X <: Q_1 \vdash S_2 <: Q_2$ 缩小成 $\Gamma, X <: T_1 \vdash S_2 <: Q_2$，这样就可以使用归纳假设了。

但是，我们需要小心一点。引理(26.4.5)表明，可以采取任意一个推导用狭窄化结论构造一个推导，但是不能保证新的推导与旧的长度保持一致。事实上，如果我们检查该引理的证明，会发现狭窄化通常产生一个更大的推导，因为每次使用 S-Tvar 来查找狭窄化的变量时，它都会在此处连接一个任意长度的推导。此外，该连接操作还将创造新的传递性规则实例，这条规则在当前的系统中是允许的。

使用基于传递性质中间类型 Q 的长度和狭窄化性质中原始边界 Q 的长度进行归纳假设，把传递性和狭窄化合起来证明，上述问题可以得到解决。

在开始证明之前，我们先来看一个引理，在上下文中置换顺序或增加新的类型变量绑定不会影响可推导的子类型声明的有效性。

28.4.1　引理[置换和弱化]：

1. 假设 Δ 是 Γ 良形式的置换[(参见(26.4.1)]，如果 $\Gamma \mapsto S <: T$，那么 $\Delta \mapsto S <: T$。
2. 如果 $\Gamma \mapsto S <: T$，并且 $dom(\Delta) \cap dom(\Gamma) = \varnothing$，则 $\Gamma, \Delta \mapsto S <: T$。

证明：常规归纳。在第(2)部分的情况 SA-All 中用到第(1)部分。

28.4.2　引理[全 $F_{<:}$ 系统的传递性和狭窄化]：

1. 如果 $\Gamma \mapsto S <: Q$ 且 $\Gamma \mapsto Q <: T$，则 $\Gamma \mapsto S <: T$。
2. 如果 $\Gamma, X <: Q, \Delta \mapsto M <: N$ 并且 $\Gamma \mapsto P <: Q$，则 $\Gamma, X <: P, \Delta \mapsto M <: N$。

证明：通过对 Q 的长度进行归纳，对两部分同时进行证明。在每一层归纳，第(2)部分假定第(1)部分讨论的 Q 已经确定。只对严格小一点的 Q，第一部分才会用到第(2)部分。

1. 我们对给定的第一个条件做内部归纳，并对两个推导的最后应用规则做情况分析。除情况 SA-All 外，其他所有的情况的证明都与引理(28.3.2)的证明一致。

   如果右端的推导是 SA-Top 实例，那么无须证明，因为 $\Gamma \vdash S <: Top$。如果左端的推导是 SA-Top 实例，则 $Q = Top$，通过观察算法规则可知右端的推导也必须是 SA-Top 实例。如果两端推导都是 SA-Refl-TVar 实例，则同样也无须证明，因为另外一个推导就是所期望得到的结论。

   如果左端的推导以 SA-Trans-TVar 实例结束，我们有 $S = Y$ 且 $Y <: U \in \Gamma, \Gamma \vdash U <: Q$，由内部归纳假设 $\Gamma \mapsto U <: T$，再根据 SA-Trans-TVar 得到 $\Gamma \mapsto Y <: T$，证毕。

   如果左端推导是以 SA-Arrow 或 SA-All 实例结束，而我们已经讨论过右端推导是 SA-Top 实例的情况，所以右端也应该是 SA-Arrow 或 SA-All 实例。如果是 SA-Arrow 情况，有 $S = S_1 \to S_2$，$Q = Q_1 \to Q_2$，$T = T_1 \to T_2$ 并且有 $\Gamma \mapsto Q_1 <: S_1$，$\Gamma \mapsto S_2 <: Q_2$，$\Gamma \mapsto T_1 <: Q_1$，$\Gamma \mapsto Q_2 <: T_2$。两次对第(1)部分应用外部归纳假设(注意 Q_1 和 Q_2 都比 Q 要小)得到 $\Gamma \mapsto T_1 <: S_1$，$\Gamma \mapsto S_2 <: T_2$，然后运用 SA-Arrow 得到 $\Gamma \mapsto S_1 \to S_2 <: T_1 \to T_2$。

   如果两个推导最后都以 SA-All 结束，有 $S = \forall X <: S_1 . S_2$，$Q = \forall X <: Q_1 . Q_2$，$T = \forall X <: T_1 . T_2$，其中子推导为：

$$\Gamma \vdash Q_1 <: S_1 \qquad \Gamma, X<:Q_1 \vdash S_2 <: Q_2$$
$$\Gamma \vdash T_1 <: Q_1 \qquad \Gamma, X<:T_1 \vdash Q_2 <: T_2$$

由外部归纳假设的第(1)部分(Q_1 比 Q 要小),结合上述两个子推导,可合并得出 $\Gamma \vdash T_1 <: S_1$。对于体,我们需要做更多工作,因为两个上下文不太一致。首先使用外部归纳假设的第(2)部分(同样需要注意 Q_1 比 Q 要小)将 $\Gamma, X <: Q_1 \vdash S_2 <: Q_2$ 中的 X 边界狭窄化,得到 $\Gamma, X <: T_1 \vdash S_2 <: Q_2$。现在应用外部归纳假设的第(1)部分($Q_2$ 比 Q 小),产生 $\Gamma, X <: T_1 \vdash S_2 <: T_2$。最后根据 SA-All,得到 $\Gamma \vdash \forall X <: S_1 . S_2 <: \forall X <: T_1 . T_2$。

2. 我们同样对第一个给定推导的长度做一下内部归纳假设,对这个推导最后用到的规则进行一下情况分析。大多数情况是直接使用内部归纳假设。最有意思的情况是 SA-Trans-TVar,令 M = X,有 $\Gamma, X <: Q, \Delta \vdash Q <: N$。进一步由内部归纳假设得到 $\Gamma, X <: P, \Delta \vdash Q <: N$。同样,对第二个给定条件应用弱化规则[引理(28.4.1),第(2)部分],得到 $\Gamma, X <: P, \Delta \vdash P <: Q$。再由第(1)部分的外部归纳假设(对同一个 Q),有 $\Gamma, X <: P, \Delta \vdash P <: N$。最后,根据 SA-Trans-TVar,得到 $\Gamma, X <: P, \Delta \vdash X <: N$,得证。

28.4.3 练习[★★★↛]:这里还有一个有关量词变式的合理的子类型化规则,比核心 $F_{<:}$ 规则更灵活,但比全 $F_{<:}$ 规则灵活性要差一点。

$$\frac{\Gamma \vdash S_1 <: T_1 \qquad \Gamma \vdash T_1 <: S_1 \qquad \Gamma, X<:T_1 \vdash S_2 <: T_2}{\Gamma \vdash \forall X<:S_1 . S_2 <: \forall X<:T_1 . T_2} \qquad \text{(S-All)}$$

这条规则近似于核心 $F_{<:}$ 变式,它不要求两个囿量词语义上相同,只要求两者等价——一个是另一个的子类型。核心规则跟这个规则的差别只在我们用一些子类型化规则产生非平凡等价类的结构,如记录来扩充语言时才表现出来,例如,在含记录的纯核心 $F_{<:}$ 系统中,类型 $\forall X <: \{a:Top, b:Top\} . X$ 不是 $\forall X <: \{b:Top, a:Top\} . X$ 的子类型。但由上面这条规则,却成了子类型。基于这条规则,子类型化是可判定的吗?

28.5 全 $F_{<:}$ 系统的不可判定性

我们在上节中说明了全 $F_{<:}$ 系统的算法子类型规则是可靠的和完备的,也就是说,这些规则下的最小闭关系与原始声明性规则下的最小闭关系包含有同样的语句。这就留给我们一个问题,是不是实现这些规则的算法对所有的输入都能终止? 遗憾的是,许多人发现(非常奇怪),回答是否定的。

28.5.1 练习[★]:如果全 $F_{<:}$ 系统算法规则定义的算法不能总是终止,那么很显然核心 $F_{<:}$ 系统的算法可终止性的证明就不能延续到全 $F_{<:}$ 系统规则中去。那么算法在那里将不再适用。

这里有个使得子类型化算法发散的例子,由 Ghelli(1995)提出。首先定义一下缩写词:

$$\neg S \quad \stackrel{\text{def}}{=} \quad \forall X<:S.X.$$

¬操作符的关键性质是其允许左右两边的子类型语句进行交换。

28.5.2　事实：$\Gamma \vdash \neg S <: \neg T$ 当且仅当 $\Gamma \vdash T <: S$。

证明：留做练习[★★↛]。

现在，我们定义类型 T：

```
T  =  ∀X<:Top. ¬(∀Y<:X.¬Y).
```

如果运用算法子类型规则自底向上来构造语句的子类型化推导：

$$X_0 <: T \ \mapsto\ X_0 \ <:\ \forall X_1 <: X_0 . \neg X_1$$

那么将会在越来越大的子目标中进行无限次回归：

$X_0<:T$	$\mapsto$	X_0	$<:$	$\forall X_1<:X_0.\neg X_1$
$X_0<:T$	$\mapsto$	$\forall X_1<:Top.\neg(\forall X_2<:X_1.\neg X_2)$	$<:$	$\forall X_1<:X_0.\neg X_1$
$X_0<:T, X_1<:X_0$	$\mapsto$	$\neg(\forall X_2<:X_1.\neg X_2)$	$<:$	$\neg X_1$
$X_0<:T, X_1<:X_0$	$\mapsto$	X_1	$<:$	$\forall X_2<:X_1.\neg X_2$
$X_0<:T, X_1<:X_0$	$\mapsto$	X_0	$<:$	$\forall X_2<:X_1.\neg X_2$
etc.				

当增加新变量时进行重命名应保持上下文良形式，应选择新的变量名称以区别回归形式。关键在于重定界，比如第二行和第三行，左端 X_1 的边界由第二行的 Top 变成第三行的 X_0。因为第二行整个左端就是 X_0 的上界，这个重定界产生了循环的模式，使得上下文中越来越长的变量链接在循环中来回移动（读者没必要对这个例子的语义进行推敲，$\neg T$ 在语法意义上为否定）。

更糟糕的是，不仅这个算法不能对某些输入终止，而且对原始全 $F_{<:}$ 系统来讲，没有一个算法是可靠且完备的，且对所有的输入能终止（Pierce，1994）。这个事实的证明对于本书而言过大，我们来看一个例子加深一下理解。

28.5.3　定义：类型 T 中正和负定义如下：在 T 中，T 本身为正。如果 $T_1 \to T_2$ 是正的（负的），那么 T_1 是负的（正的），T_2 是正的（负的）。如果 $\forall X <: T_1 . T_2$ 是正的（负的），则 T_1 是负的（正的），T_2 是正的（负的）。子类型声明 $\Gamma \vdash S <: T$ 中的正和负定义如下：类型 S 和上下文 Γ 中类型变量的边界是负的，类型 T 是正的。

“正”和“负”出自逻辑学。根据著名的命题与类型 Curry-Howard 对应理论（参见 9.4 节），类型 $S \to T$ 对应于 $S \Rightarrow T$，而由逻辑蕴涵的定义，与 $\neg S \vee T$ 等价。子命题 S 显然是否定的，也就是说 S 包含奇数个否定，当且仅当整个蕴涵包含偶数个否定。注意：出现 T 的正对应于 $\neg T$ 的负。

28.5.4　事实：如果 X 在 S 中是正的，在 T 中是负的，则 $X <: U \vdash S <: T$，当且仅当 $\vdash [X \mapsto U]S <: [X \mapsto U]T$。

证明：留做练习[★★ ↛]

现在：

```
T  =  ∀X0<:Top.∀X1<:Top.∀X2<:Top.
         ¬(∀Y0<:X0.∀Y1<:X1.∀Y2<:X2.¬X0)
```

我们来看下面这个子类型语句：

```
⊢   T   <:   ∀X0<:T.∀X1<:P.∀X2<:Q.
               ¬(∀Y0<:Top.∀Y1<:Top.∀Y2<:Top.
                 ¬(∀Z0<:Y0.∀Z1<:Y2.∀Z2<:Y1.U)).
```

可以把上述语句比做对计算机状态的描述。变量 X_1 和 X_2 是寄存器，它们目前存放的内容是类型 P 和 Q。机器的指令流是第三行：第一条指令在 Z_1 和 Z_2 的范围内（Y_2 和 Y_1，注意顺序）进行编码，未说明的类型 U 代表程序中其余的指令。类型 T，嵌套的否定，以及囿变量 X_0 和 Y_0 具有同样的功能：作为上述简单例子的副本，它们允许我们转动曲柄，回到相同形状的子目标，作为我们的初始目标。曲柄的一次转动代表机器的一次轮转。

在这个例子中，在指令流中处于前端的指令编码成这个命令：转变寄存器 1 和 2 中的内容。用上面陈述的两条事实做如下计算（为便于分析值 P 和 Q 都用阴影显示）：

```
         ⊢   T
         <:  ∀X0<:T.∀X1<:P.∀X2<:Q.
               ¬(∀Y0<:Top.∀Y1<:Top.∀Y2<:Top.
                 ¬(∀Z0<:Y0.∀Z1<:Y2.∀Z2<:Y1.U))
当且仅当 ⊢   ¬(∀Y0<:T.∀Y1<:P.∀Y2<:Q.¬T)
         <:  ¬(∀Y0<:Top.∀Y1<:Top.∀Y2<:Top.                 根据事实(28.5.4)
               ¬(∀Z0<:Y0.∀Z1<:Y2.∀Z2<:Y1.U))
当且仅当 ⊢   (∀Y0<:Top.∀Y1<:Top.∀Y2<:Top.
               ¬(∀Z0<:Y0.∀Z1<:Y2.∀Z2<:Y1.U))
         <:  (∀Y0<:T.∀Y1<:P.∀Y2<:Q.¬T)                      根据事实(28.5.2)
当且仅当 ⊢   ¬(∀Z0<:T.∀Z1<:Q.∀Z2<:P.U))
         <:  ¬T                                              根据事实(28.5.4)
当且仅当 ⊢   T
         <:  (∀Z0<:T.∀Z1<:Q.∀Z2<:P.U))                      根据事实(28.5.2)
```

注意到，在推导的最后，不仅 P 和 Q 交换了位置，而且导致这个动作发生的那条指令也执行过了，让其在指令流的前端 U 下一个执行。按刚刚执行指令的相同方式来开始选择 U 的值：

```
U   =   ¬(∀Y0<:Top.∀Y1<:Top.∀Y2<:Top.
          ¬(∀Z0<:Y0.∀Z1<:Y2.∀Z2<:Y1.U'))
```

在继续使用 U′以前，我们可以进行另外一次交换，把寄存器返回到它们原始的状态。可选择地，我们对 U 选择不同的值来进行不同的操作。例如，如果：

```
U   =   ¬(∀Y0<:Top.∀Y1<:Top.∀Y2<:Top.
          ¬(∀Z0<:Y0.∀Z1<:Y1.∀Z2<:Y2.Y1))
```

然后，在机器的下一个周期，寄存器 1 的当前值 Q，将出现在 U 的位置上，这样做其实是通过 Q 所表示的指令流的寄存器 1 来实现一个“间接”分支。条件构造子和算术（前驱，后继以及零测试）都可以由这种方法来实现。

综上所述，我们可以通过从双计数器机器（普通图灵机上的一个简单变式，包括一个有限的控制和两个计数器，每个计数器中存放着一个自然数）到子类型化语句进行归约中得出不可判定的证明。

28.5.5　定理［Pierce，1994］：对于每一个双计数器的机器 M，存在一个子类型语句 S(M)在全 $F_{<:}$ 系统中是可推导的，当且仅当 M 的执行可终止。

因此，如果我们可以确定一个子类型语句是不是可证明，那么同样可以确定一个任意给定的双计数器——机器是否会停机。因为双计数器停机问题具有不可判定性（参考 Hopcroft 和 Ullman，1979），所以对于全 $F_{<:}$ 系统的子类型化同样也是不可判定的。

我们必须再次强调一下，子类型关系的不可判定性并不意味着我们在 28.4 节中讨论的子类型化的半算法是不可靠和不完备的。根据声明性的子类型化规则，如果 $\Gamma \vdash S <: T$ 是可证明的，则算法必然可以终止并且结果为 true。若根据声明性子类型化规则 $\Gamma \vdash S <: T$ 是不可证明的，那么算法将会发散或者结果为 false。一个给定的子类型语句以两种不同的方式，不能从算法规则中证明：产生一个子目标的无穷序列（意味着该结论没有一个有限的推导）或者导致一个明显不一致，如 $Top <: S \to T$。子类型算法只能检测到其中之一，而不能检测到另一个。

全 $F_{<:}$ 系统的不可判定性是否就意味着该系统在实际中就没有用处了呢？事实上，我们通常认为，全 $F_{<:}$ 系统的不可判定性不是一个非常严重的缺陷。一方面，有资料表明（Ghelli，1995）要使子类型检查器发散，它必须具有三个特殊性质，每一性质都不可能由程序员碰巧创造出来。同样地，现在有一些流行的语言，其类型检查和类型重构问题或者解决耗费巨大代价，比如在 22.7 节中看到的 ML 和 Haskell，或者不可判定，比如 C++ 和 λProlog（Felty，Gunter，Hannan，Miller，Nadathur 和 Scedrov，1988）。事实上，经验告诉我们，缺乏交类型和并类型（下节将做讨论）（参见练习 28.6.3）对于全 $F_{<:}$ 系统来说会带来比不可判定性更严重的缺陷。

28.5.6　练习[★★★★]：(1)用 $X <: T$ 和 X 变量绑定（囿量词或非囿量词都可以）而不是用 Top 类型来定义全 $F_{<:}$ 系统的一个变式，这个变式称为完全囿量词；(2)证明该系统的子类型关系是可判定的；(3)这条限制是否对本节中的基本问题提供了一个令人满意的解决方案？特别地，它是否也适用于那些带有数字、记录、变式等附加特征的语言？

28.6　合类型和交类型

在 16.3 节中，我们看到，带子类型化语言中一个非常可取的特性是每个类型 S 和 T 序对都存在合类型，即 S 和 T 共同超类的最小类型 J。在本节中，我们将说明核心 $F_{<:}$ 系统中的子类型关系对每一个 S 和 T 的确有一个合类型，以及对只要给出了计算的算法，每一个 S 和 T 至少含共同的子类型交类型（另一方面，所有的这些特性对全 $F_{<:}$ 系统都不适用，参见练习 28.6.3）。

用 $\Gamma \vdash S \vee T = J$ 表示 J 是上下文 Γ 中 S 和 T 的合类型，$\Gamma \vdash S \wedge T = M$ 表示 M 是上下文 Γ 中 S 和 T 的交类型。它们的计算规则同时也定义了（如图 28.5 所示）。注意，在每个定义中有些情况是重复的，为选择一个作为明确算法的定义，这里规定选择第一条子句。

很容易检查到 $\vee$ 和 $\wedge$ 是全函数，因为 $\vee$ 通常返回一个类型而 $\wedge$ 要么返回某个类型，要么操作失败。我们观察到 Γ 中 S 和 T 的总权值[参见定义(28.3.4)]在递归调用中是逐渐减小的。

现在我们来验证一下这些定义确实计算了合类型和交类型。论证过程分为两部分：命题(28.6.1)表明合类型是 S 和 T 的上界，而交类型（如果存在的话）是下界。命题(28.6.2)表明合类型比 S 和 T 的所有共同上界都要小，而交类型比 S 和 T 的所有共同的下界都要大（只要 S 和 T 含有共同下界它就存在）。

$\rightarrow \forall$ <: Top

$$\Gamma \vdash S \vee T = \begin{cases} T & \text{if } \Gamma \vdash S <: T \\ S & \text{if } \Gamma \vdash T <: S \\ J & \text{if } S = X \\ & \quad X<:U \in \Gamma \\ & \quad \Gamma \vdash U \vee T = J \\ J & \text{if } T = X \\ & \quad X<:U \in \Gamma \\ & \quad \Gamma \vdash S \vee U = J \\ M_1 \rightarrow J_2 & \text{if } S = S_1 \rightarrow S_2 \\ & \quad T = T_1 \rightarrow T_2 \\ & \quad \Gamma \vdash S_1 \wedge T_1 = M_1 \\ & \quad \Gamma \vdash S_2 \vee T_2 = J_2 \\ \forall X<:U_1.J_2 & \text{if } S = \forall X<:U_1.S_2 \\ & \quad T = \forall X<:U_1.T_2 \\ & \quad \Gamma, X<:U_1 \vdash S_2 \vee T_2 = J_2 \\ Top & \text{其他} \end{cases}$$

$$\Gamma \vdash S \wedge T = \begin{cases} S & \text{if } \Gamma \vdash S <: T \\ T & \text{if } \Gamma \vdash T <: S \\ J_1 \rightarrow M_2 & \text{if } S = S_1 \rightarrow S_2 \\ & \quad T = T_1 \rightarrow T_2 \\ & \quad \Gamma \vdash S_1 \vee T_1 = J_1 \\ & \quad \Gamma \vdash S_2 \wedge T_2 = M_2 \\ \forall X<:U_1.M_2 & \text{if } S = \forall X<:U_1.S_2 \\ & \quad T = \forall X<:U_1.T_2 \\ & \quad \Gamma, X<:U_1 \vdash S_2 \wedge T_2 = M_2 \\ \textit{fail} & \text{其他} \end{cases}$$

图 28.5 核心 $F_{<:}$ 系统合类型和交类型规则

28.6.1 命题:

1. 如果 $\Gamma \vdash S \vee T = J$,则有 $\Gamma \vdash S <: J$ 和 $\Gamma \vdash T <: J$。
2. 如果 $\Gamma \vdash S \wedge T = M$,则有 $\Gamma \vdash M <: S$ 和 $\Gamma \vdash M <: T$。

证明:对推导 $\Gamma \vdash S \vee T = J$ 或 $\Gamma \vdash S \wedge T = M$ 的长度进行直接归纳(即应用递归调用的次数来计算 J 和 M)。

28.6.2 命题:

1. 如果 $\Gamma \vdash S <: V$ 和 $\Gamma \vdash T <: V$,则对某些 J,满足 $\Gamma \vdash J <: V$,存在 $\Gamma \vdash S \vee T = J$。
2. 如果 $\Gamma \vdash L <: S$ 和 $\Gamma \vdash L <: T$,则对某些 M,满足 $\Gamma \vdash L <: M$,存在 $\Gamma \vdash S \wedge T = M$。

证明:对第(1)部分的 $\Gamma \mapsto S <: V$ 和 $\Gamma \mapsto T <: V$ 及第(2)部分 $\Gamma \mapsto L <: S$ 和 $\Gamma \mapsto L <: T$ 的算法推导的总长同时直接进行归纳就很容易证明出来[参见定理(28.3.3)保证,这类给定推导的副本总是存的]。

1. 如果两个推导都是 SA-Top 类型的实例,V = Top,则立即可以得到期望的结果 $\Gamma \vdash J <: V$。
   如果 $\Gamma \mapsto T <: V$ 是 SA-Refl-TVar 类型实例,则 T = V。由给定的第一个推导有 $\Gamma \vdash S <: V = T$,所以可运用合类型定义中的第一条句子得到 $\Gamma \vdash S \vee T = T$,满足要求。类似地,如果 $\Gamma \vdash S <: V$ 是 SA-Refl-TVar 类型实例,则 S = V。由给定的第二个推导有 $\Gamma \vdash T <: V = S$,运用合类型定义的第二条句子得到 $\Gamma \vdash S \vee T = S$,满足要求。
   如果在推导 $\Gamma \mapsto S <: V$ 以规则 SA-Trans-TVar 为结束,则有 S = X,其中 $X <: U \in \Gamma$ 以及 $\Gamma \vdash U \vee T = J$,再由合类型定义的第三句,得到 $\Gamma \vdash S \vee T = J$,根据归纳假设得到 $\Gamma \vdash J <: V$。推导 $\Gamma \mapsto T <: V$ 以规则 SA-Trans-TVar 结束的情况类似。
   从算法子类型规则的形式上看,容易推断剩下的可能性就是两个推导条件不是以 SA-Arrow,就是以 SA-All 规则为结束的。

如果两个推导的最后规则都是 SA-Arrow,则有 $S = S_1 \to S_2$, $T = T_1 \to T_2$ 和 $V = V_1 \to V_2$,有 $\Gamma \mapsto V_1 <: S_1$, $\Gamma \mapsto S_2 <: V_2$, $\Gamma \mapsto V_1 <: T_1$ 及 $\Gamma \mapsto T_2 <: V_2$。由对第(2)部分的归纳假设有 $\Gamma \vdash S_1 \wedge T_1 = M_1$,存在某 M_1 满足 $\Gamma \vdash V_1 <: M_1$。而由第(1)部分,$\Gamma \vdash S_2 \vee T_2 = J_2$,存在某 J_2 满足 $\Gamma \vdash J_2 <: V_2$。再由合类型定义的第 5 条句子得到 $\Gamma \vdash S_1 \to S_2 \vee T_1 \to T_2 = M_1 \to J_2$,并且根据 SA-Arrow,得到 $\Gamma \vdash M_1 \to J_2 <: V_1 \to V_2$。

最后,如果两个推导的最后规则都是 SA-All,那么有 $S = \forall X <: U_1 . S_2$, $T = \forall X <: U_1 \to T_2$ 和 $V = \forall X <: U_1 . V_2$,其中 $\Gamma, X <: U_1 \mapsto S_2 <: V_2$ 且 $\Gamma, X <: U_1 \mapsto T_2 <: V_2$。由第(1)部分的归纳假设,有 $\Gamma, X <: U_1 \vdash S_2 \vee T_2 = J_2$,其中 $\Gamma, X <: U_1 \mapsto J_2 <: V_2$,根据合类型定义的第 6 句,有 $J = \forall X <: U_1 . J_2$。再根据 SA-All 得到 $\Gamma \vdash \forall X <: U_1 . J_2 <: \forall X <: U_1 . V_2$。

2. 如果推导 $\Gamma \mapsto L <: T$ 的最后规则是 SA-Top,则 $T = Top$,所以 $\Gamma \vdash S <: T$。由交类型定义的第一句,我们有 $\Gamma \vdash S \wedge T = S$,而根据另外一个推导有 $\Gamma \vdash L <: S$,得证。推导 $\Gamma \mapsto L <: S$ 的最后的规则是 SA-Top 的证明类似。

 如果推导 $\Gamma \mapsto L <: S$ 的最后的规则是 SA-Refl-TVar,则 $L = S$,由另一个推导有 $\Gamma \vdash L = S <: T$,由交类型定义有 $\Gamma \vdash S \wedge T = S$,由此得证。推导 $\Gamma \mapsto L <: T$ 的最后的规则是 SA-Refl-TVar,则证明类似。

 剩下的可能性就只有两个给定推导的最后规则为 SA-Trans-TVar, SA-Arrow 或 SA-All 的情况了。

 如果两个推导的最后规则是 SA-Trans-TVar,我们有 $L = X$,其中 $X <: U \in \Gamma$,并且有 $\Gamma \mapsto U <: S$ 和 $\Gamma \mapsto U <: T$。由第(2)部分的归纳假设,$\Gamma \vdash U <: M$,由此再根据 S-TVar 和传递性,得到 $\Gamma \vdash L <: M$。

 如果两个推导的最后规则是 SA-Arrow,有 $S = S_1 \to S_2$, $T = T_1 \to T_2$, $L = L_1 \to L_2$,其中 $\Gamma \mapsto S_1 <: L_1$, $\Gamma \mapsto L_2 <: S_2$, $\Gamma \mapsto T_1 <: L_1$ 及 $\Gamma \mapsto L_2 <: T_2$。由第(1)部分的归纳假设,存在某个 J_1,满足 $\Gamma \vdash J_1 <: L_1$,使得 $\Gamma \vdash S_1 \vee T_1 = J_1$。由第(2)部分,存在某个 M_2,满足 $\Gamma \vdash L_2 <: M_2$,使得 $\Gamma \vdash S_2 \wedge T_2 = M_2$。再由交类型的定义有 $\Gamma \vdash S_1 \to S_2 \wedge T_1 \to T_2 = J_1 \to M_2$,再根据 S-Arrow,我们得到 $\Gamma \vdash L_1 \to L_2 <: J_1 \to M_2$。

 两个推导的最后规则都是 SA-All 的情况的证明类似。

28.6.3 练习[推荐,★★★]: 考虑下面两个类型(Ghelli, 1990) $S = \forall X <: Y \to Z . Y \to Z$ 和 $T = \forall X <: Y' \to Z' . Y' \to Z'$,以及上下文 $\Gamma = Y <: Top, Z <: Top, Y' <: Y, Z' <: Z$。(1)在全 $F_{<:}$ 系统中,有多少类型在上下文 Γ 下既是 S 的子类型又是 T 的子类型;(2)证明在全 $F_{<:}$ 系统中,S 和 T 没有交类型;(3)在全 $F_{<:}$ 中找到一类型序对,使其在上下文 Γ 中没有合类型。

28.7 囿存在量词

为了把核心 $F_{<:}$ 系统类型检查算法扩展到带存在类型的语言中,我们还得处理另外一个小问题。回忆一下下面囿量词的消去规则:

$$\frac{\Gamma \vdash t_1 : \{\exists X <: T_{11}, T_{12}\} \qquad \Gamma, X <: T_{11}, x : T_{12} \vdash t_2 : T_2}{\Gamma \vdash \texttt{let}\ \{X, x\} = t_1\ \texttt{in}\ t_2 : T_2} \quad \text{(T-Unpack)}$$

在 24.1 节中,我们注意到类型变量 X 出现在上下文中第二个前提 t_2 的类型被计算的位置,而不是出现在规则结论的上下文中。这意味着类型 T_2 不可能含自由的 X,因为结论中任何自由变量 X 的出现都将是越界的。25.5 节对这个问题进行了更加详细的讨论,我们还注意到,当使用无名 deBruijn 格式来描述类型时,上下文中从前提到结论的变化对应了 T_2 中变量索引的一个负的移位;而如果 T_2 碰巧含有自由变量 X 则这种移位将失败。

对带存在量词语言的最小类型算法这意味着什么? 特别地,我们应该怎样处理表达式 t = let{X,x} = p in x,其中 p 类型为{∃X,Nat→X}? x 最自然的类型是 Nat→X,X 为囿变量。但是,根据声明性类型关系(带包含规则),x 同样具有类型 Nat→Top 和 Top。因为这两个都没有涉及 X,在声明性系统中整个项 t 可以被合法地赋予类型 Nat→Top 或 Top。更一般地,我们通常自由地把一个未打包的表达式提升成任意一个不含囿变量 X 的类型,然后再运用 T-Unpack。所以,如果我们要使这个最小类型算法具有完备性,则当未打包的表达式中最小类型 T_2 包含界变量 X 时,不会简单地失败。而是尽力提升 T_2 为不含囿变量 X 的超类型。关键是给定类型的不含囿变量 X 的超类型集有个最小的元素,如下面这个练习(答案由 Ghelli 和 Pierce,1998 给出)。

28.7.1 练习[★★★]:在带囿存在量词的核心 $F_{<:}$ 系统中,给出一个算法,计算有关上下文 Γ 中给定类型 T 的不含囿变量 X 的最小超类型,记为 $R_{X,\Gamma}(T)$。

存在量词消去的算法类型规则可以写为:

$$\frac{\Gamma \mapsto t_1 : T_1 \qquad \Gamma \mapsto T_1 \Uparrow \{\exists X{<:}T_{11}, T_{12}\} \qquad \Gamma, X{<:}T_{11}, x{:}T_{12} \mapsto t_2 : T_2 \qquad R_{X,(\Gamma, X<:T_{11}, x:T_{12})}(T_2) = T_2'}{\Gamma \mapsto \texttt{let } \{X,x\}{=}t_1 \texttt{ in } t_2 : T_2'} \quad \text{(TA-Unpack)}$$

对带囿存在量词的全 $F_{<:}$ 系统而言,这种情况的问题更大。Ghelli 和 Pierce(1998)举了一个例子,上下文 Γ 中不含囿变量 X 超类型 T 没有最小的元素。这也立即说明了该系统的类型关系缺少最小类型的特点。

28.7.2 练习[★★★]:证明只含囿存在量词(不是全称类型)的全 $F_{<:}$ 系统中变式类型的子类型关系同样是不可判定的。

28.8 囿量词和最小类型

最小类型 Bot(参见 15.4 节)在某种程度上使得 $F_{<:}$ 的元理论属性复杂化了。原因是,在形如 ∀X <: Bot.T 的类型中,变量 X 实际上是 T 中 Bot 的同义词,因为根据假设,X 是 Bot 的子类型,而由 S-Bot,Bot 又是 X 的子类型。这就意味着 ∀X <: Bot.X→X 和 ∀X <: Bot.Bot→Bot 在子类型关系下是等价的,尽管它们在语法上并不相同。另外,如果周围的环境包含假定 X <: Bot 以及 Y <: Bot,则类型 X→Y 和类型 Y→X 是等价的,即使它们都没有明确地提及 Bot。尽管有这些困难存在,核心 $F_{<:}$ 的那些关键性质仍然可以在 Bot 存在的情况下确定下来。详情参见 Pierce(1997a)。

第六部分　高阶系统

第 29 章　类型算子和分类[①]

在以前的章节中，我们通常使用缩写形式，比如：

```
CBool = ∀X. X → X → X;
Pair Y Z = ∀X. (Y→Z→X) → X;
```

使例子更易阅读，如把繁琐形式 λx: ∀X.(Nat→Bool→X)→X.x 写做 λx:Pair Nat Bool.x。

CBool 是一个简单的缩写，当它出现在一个例子中时，我们需要用其定义的右端形式对其进行替换。Pair 是一个参数性缩写，当使用 Pair S T，必须使用 S 和 T 分别对 Pair 定义中的 Y 和 Z 进行替换。换句话说，像 Pair 的这种缩写给了我们一个类型表达式级别函数的非正式定义。

我们也使用诸如 Array T 和 Ref T 这样的包含类型算子 Array 和 Ref 的类型表达式。尽管这些类型算子是在语言中构造的，而不是由程序员定义的，它们在类型的级别上仍然是一种函数。例如，我们可以把 Ref 视为这样一种函数，对于每一个类型 T，产生一个可以包含 T 类型元素的引用单元。

在本章和接下来的两章中，我们将对这些类型级别的函数进行讨论，称为类型算子，这样会更正式一点。在本章中，我们将在类型的层次上介绍抽象和应用的基本机制，并将给出两个类型表达式在何时是等价的和为良形式关系的一种精确定义，称为分类。分类可以使我们避免写那些无意义的类型表达式。第 30 章将进一步讨论把类型算子当做“一等公民”，即实体，作为参数传递给函数。第 30 章还介绍了著名的 F_ω 系统，把 F 系统（参见第 23 章）中类型量词概化成类型算子的高阶量词。第 31 章将对类型算子，高阶量词以及子类型化的联合进行讨论。

29.1　直觉

要研究类型级别的函数，我们需要做的第一件事情就是确定抽象和应用中用到的符号。标准惯例是使用与项层次上的应用和抽象相同的符号，用 λ 表示抽象，用并列形式符表示应用[②]。比如，函数 λX.{a:X,b:X}，给定一个类型 T，构造出一个记录类型{a:T,b:T}。将函数应用在参数 Bool 上记为(λX.{a:X,b:X})Bool。

与普通的函数一样，多参数的类型函数可以由单参数函数通过 Currying 化来建立。例如，类型表达式 λY.λZ. ∀X.(Y→Z→X)→X 有两个参数的函数，严格地讲，其具有一个参数的函数，当把该函数应用到类型 S 中时，产生了另外一个单参数函数。当把新函数应用到 T 时，产生类型 ∀X.(S→T→X)→X。

---

① 本章介绍的系统为纯简单类型化 λ 演算，其中含类型算子 λ_ω（参见图 29.1）。本章举的例子也用了数字型和布尔型（参见 8.2 节）及全称类型（参见图 23.1）。相关的 OCaml 实现为 fullomega。

② 这种过于简单记法的一个缺点是使得不同种类表达式的术语有些扭曲。特别地，“类型抽象”现在也许就表示一种把类型看做参数的抽象（比如，项 λX.t），或者说相当于类型层次上的抽象（比如，类型表达式 λX.{a:X}）。在上下文中，两者都是可能的，人们倾向于用多态函数来表示前者，用类型层次上的抽象或者算子抽象来表示后者。

我们将继续使用非正式的缩写代替长的类型表达式，包括类型算子。例如，在本章的其余部分将使用下面这个缩写：

```
Pair = λY. λZ. ∀X. (Y→Z→X) → X;
```

当我们使用 Pair S T，实际的含义为：

```
(λY. λZ. ∀X. (Y→Z→X) → X) S T.
```

换句话说，我们把这里用到的参数式缩写非正式惯例替换为更加基本的非正式惯例，即当看到这些缩写形式，就用它们定义的右端对简单缩写进行扩展，加上类型算子定义和实例化的正规机制。我们可以正规地处理缩写的定义和扩展操作，比如，我们可以在对象语言中完成这些操作，而不是在元语言中作为约定来实现，但这里我们不这样做。感兴趣的读者可以翻阅有关含定义或单点分类的类型系统的文献；参见 Severi 和 Poll(1994)，Stone 和 Harper(2000)，Crary (2000)，以及其他文献。

在类型的层次上介绍抽象和应用使得我们可以用多种不同的方式写同一个类型。例如，如果 Id 是类型算子 λX.X 的缩写，则下列表达式均为相同箭头类型的名称：

```
Nat → Bool        Nat → Id Bool        Id Nat → Id Bool
Id Nat → Bool     Id (Nat → Bool)      Id (Id (Id Nat → Bool))
```

为了使这个直觉更加精确，我们引入类型的定义性等价关系，记为 S ≡ T。这个关系的定义中最重要的语句是：

$$(\lambda X{::}K_{11}.T_{12})\ T_2 \equiv [X \mapsto T_2]T_{12} \qquad \text{(Q-AppAbs)}$$

该句说明，应用到参数的类型层次上的抽象与代换为形参的参数抽象体是等价的。我们用一条新规则 T-Eq：

$$\frac{\Gamma \vdash t : S \qquad S \equiv T}{\Gamma \vdash t : T} \qquad \text{(T-Eq)}$$

对类型检查中的定义性等价进行了探讨，得出一结论：如果两个类型是等价的，那么一个类型的成员也是另一个类型的成员。

抽象和应用机制的另外一种新的可能性就是可以写出没有意义的类型表达式。例如，把某个特有的类型应用到另外一个类型当中去，如类型表达式(Bool Nat)，这与项层次上把 true 应用到 6 中去一样没有意义。为了避免这类无意义的表达式的出现，我们引入了分类系统，按照表达式的元素对其进行分类，就如同箭头类型告诉我们项的元素。

分类是从一个单一的原子分类构造起来的，记做“ * ”，读做“类型”，以及一个类型构造子 ⇒。它们包括：

*	特有类型的分类(比如 Bool 和 Bool→Bool)
* ⇒ *	类型算子的分类(类型到类型的函数)
* ⇒ * ⇒ *	特有类型到类型算子的函数分类(双目运算符)
(* ⇒ *)⇒ *	类型算子到特有类型的函数分类

分类即“类型的类型”。实质上，分类系统是简单类型的 lambda 演算的一个副本，只不过高了一层。

为了保持一致，我们仍然把任何类型层次的表达式称为类型，不管其是诸如 Nat→Nat 和 ∀X.X→X 的普通类型，还是诸如 λX.X 的类型算子。当专门对普通类型（实际用来对项进行分类的类型表达式）进行讨论时，我们称其为合适类型。

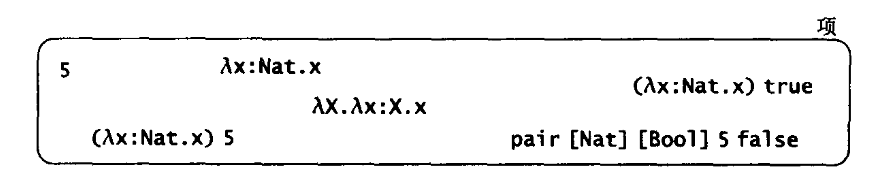
形如（$*\Rightarrow*$）$\Rightarrow*$ 的类型表达式称为高阶类型算子。不像项层次上的高阶函数那样非常有用，高阶类型算子有些深奥。我们将在第 32 章来看一些使用高阶类型算子的例子。

为了简化类型表达式良分类检查问题，我们对每一个类型层次上的抽象用其囿变量的分类做注释。例如，Pair 算子的正式形式为：

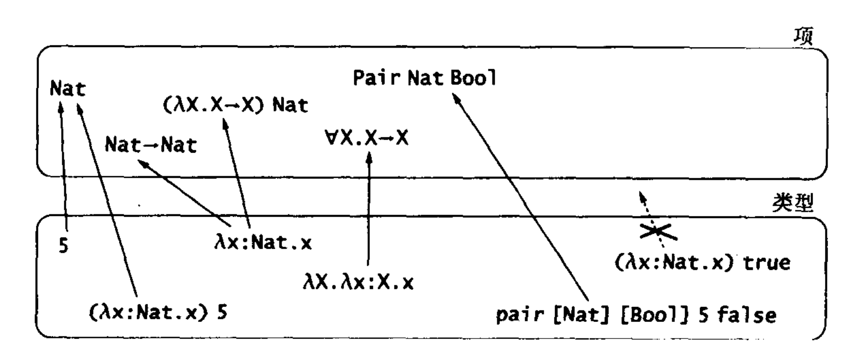
```
Pair = λA::*. λB::*. ∀X.  (A→B→X) → X;
```

（注意双冒号）然而，几乎所有的注释都是 $*$，所以我们继续将 λX.T 作为 λX::$*$.T 的缩写。

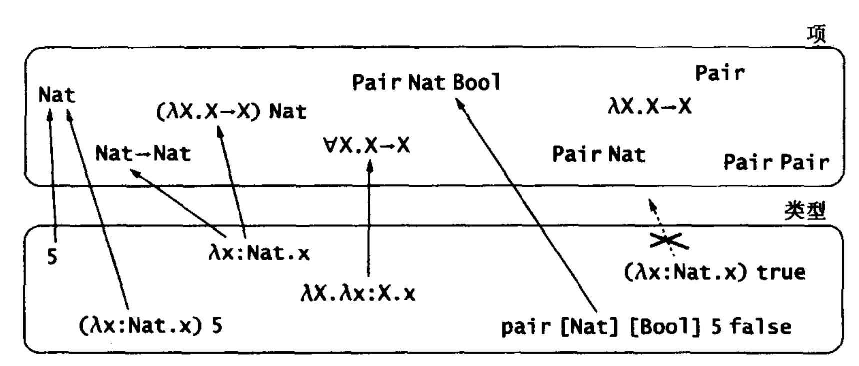
下面一些图片将使我们理解得更加清晰。语言中的表达式可以划分成三类：项、类型和分类。项包括基本数值（整型、浮点型）、复合数据（记录）、数值抽象、应用、类型抽象和类型应用，等等。



类型层次上包括两类表达式，一类是形如 Nat，Nat→Nat，Pair Nat Bool，以及 ∀X.X→X 的合适类型，项依据这些类型产生［当然，并不是所有的项都属于某个类型；(λx:Nat.x)true 就不是任何类型］。

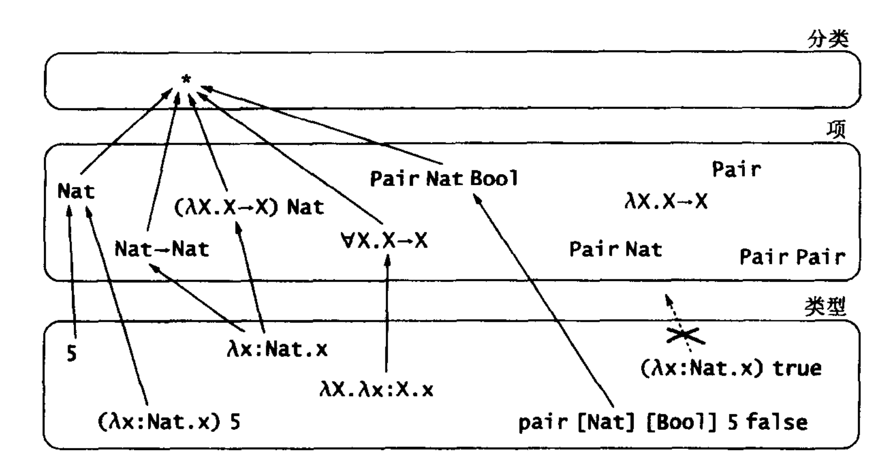


还有一类是类型算子，比如 Pair 和 λX.X→X，它们本身是不对项进行分类的（问"什么项具有类型 λX.X→X"这种问题是没有意义的），但是给它们一个类型参数生成某个合适类型后，比如 (λX.X→X)Nat，就可以对项进行分类了。

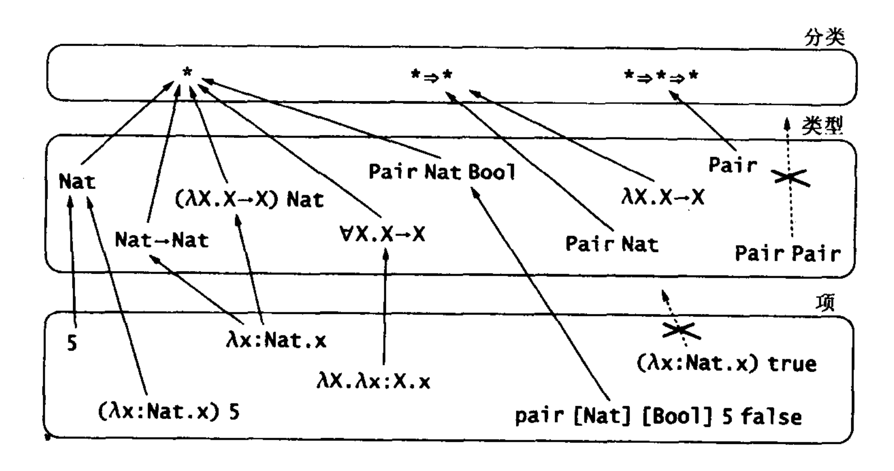


注意,合适类型;也就是分类 * 的类型表达式,可能包含比其分类还要高的类型算子作为其子表达式,比如在(λX.X→X)Nat 或 Pair Nat Bool。类似地,基类型如 Nat 的项表达式也可能包含 lambda 抽象作为其子表达式,比如(λx:Nat.x) 5。

最后,我们来看一下分类层次。最简单的分类是 * ,所有的合适类型都是其成员。



类型算子 λX.X→X 和 Pair 属于箭头分类如 * ⇒ * 和 * ⇒ * ⇒ * 。不良格式的类型表达式,比如 Pair Pair 不属于任何分类。



29.1.1　练习[★]:类型表达式 ∀X.X→X 与 λX.X→X 有什么差别?

29.1.2　练习[★]:为什么箭头类型 Nat→Nat 不属于箭头分类 * ⇒ * ?

在分层问题上,一个很自然的问题就是,“为什么表达式就只分这三层?”。我们不能继续引入从分类到分类的函数或者分类层次上的应用吗?或增加第四层按其功能性对分类表达式进行分类?依次无穷下去。纯类型系统研究组曾对这种系统做过研究(Terlouw, 1989; Berardi, 1988, Barendregt, 1991, 1992; Jutting, McKinna 和 Pollack, 1994; McKinna 和 Pollack, 1993; Pollack, 1994)。对编程语言来讲,这三层已经足够了。

事实上,虽然在大多数静态类型编程语言中都能发现某种形式的类型算子,但语言设计者给程序员提供形式化说明的全部功能的情况是很少的。有些语言(比如 Java)仅仅提供了一些固定的类型算子,比如 Array,而且不能定义新的算子。其他一些语言把类型算子和该语言的

其他一些特点绑定起来。在 ML 语言中,类型算子是数据类型机制的一部分,我们可以按如下方法定义参数性数据类型[①]:

```
type 'a Tyop = tyoptag of ('a → 'a);
```

其中将 Tyop 写为:

```
Tyop = λX. <tyoptag:X→X>;
```

换句话说,在 ML 语言中,我们可以定义参数性变式类型,而不能任意定义参数类型。这条限制的好处在于,在类型层次上,程序中出现 Tyop 的地方,对应于项层次上标记 tyoptag 将出现,即当类型检查器需要用定义的等价关系把类型 Tyop Nat 替换成其归约形式 Nat→Nat 时,就可以在程序中明确地标注 tyoptag。这从本质上简化了类型检查算法[②]。

分类构造子⇒是这里惟一讨论的构造子,但我们对其他许多分类构造子也曾做过研究。事实上,对类型表达式的不同性质进行检查和追踪的分类系统的范围跟对项的属性进行分析的类型系统的范围是对立的。有以下几类分类存在,记录分类(它的元素是类型的记录,不要与记录类型混淆起来;它们提供了一种自然的方式来定义互递归类型系统),行分类(在具有行变量多态性的系统中,行分类描述了可以构造记录类型的字段的行,参 22.8 节),power 分类或 power 类型(提供了子类型化的另外一种表示方式,参见 Cardelli,1988a),单个分类(与定义有关,以及与带共享的模块系统有关),依赖分类(与在 30.5 节中讨论的依赖类型的上一级相似)等。

29.2 定义

图 29.1 列出了带类型算子的核心 lambda 演算的完整定义。在项一级,简单类型 lambda 演算仅包含变量、抽象和应用(就是因为这个原因,才称之为带类型算子的简单类型 lambda 演算)。在类型一级,包括普通的箭头类型、类型变量、外加算子抽象和应用。在本章讨论的这个系统中忽略了诸如 $\forall X.T$ 的量词类型,我们将在第 30 章中对量词类型重新进行讨论。

这个系统的表示以三种方式对简单类型 lambda 演算的框架做了扩展。第一,增加了一组分类规则,详细说明了如何组合一些类型表达式来产生新的类型表达式。$\Gamma\vdash T::K$ 表示"类型 T 在上下文 Γ 中有分类 K"。注意这些分类规则和原始的简单类型 lambda 演算的分类规则(参见图 9.1)的相似性。

第二,当类型 T 出现在项中(比如在 $\lambda x.T.t$),我们必须检查一下 T 是不是格式良好的。这包括对老规则 T-Abs 增加一条新的前提以检查 $\Gamma\vdash T::*$。注意,T 必须有确切的分类 *,也就是说 T 必须是合适类型,因为它将用来描述项变量 x 涉及到的值。类型规则保持不变,只要 $\Gamma\vdash t:T$ 是可推导的,则 $\Gamma\vdash T::*$ 也是可推导的(只要上下文 Γ 中出现的这些类型是良分类的)。我们将在 30.3 节中对这个问题做更详细的讨论。

---

① 从例子的角度,我们忽略了 ML 语言中标识符以大写字母开头的习惯。在 OCaml 中,这个定义写成:

type'a tyop = Tyoptag of ('a→'a)

② 这条限制类似于 ML 语言中对递归类型的处理,我们在 20.1 节曾经讨论过。把递归类型绑定到 datatype 定义中去,使得程序员可以方便地使用相等递归类型,使类型检查器通过在标记和与变式类型有关的情况分析操作中隐藏 fold/unfold 注释简化了同构类型。

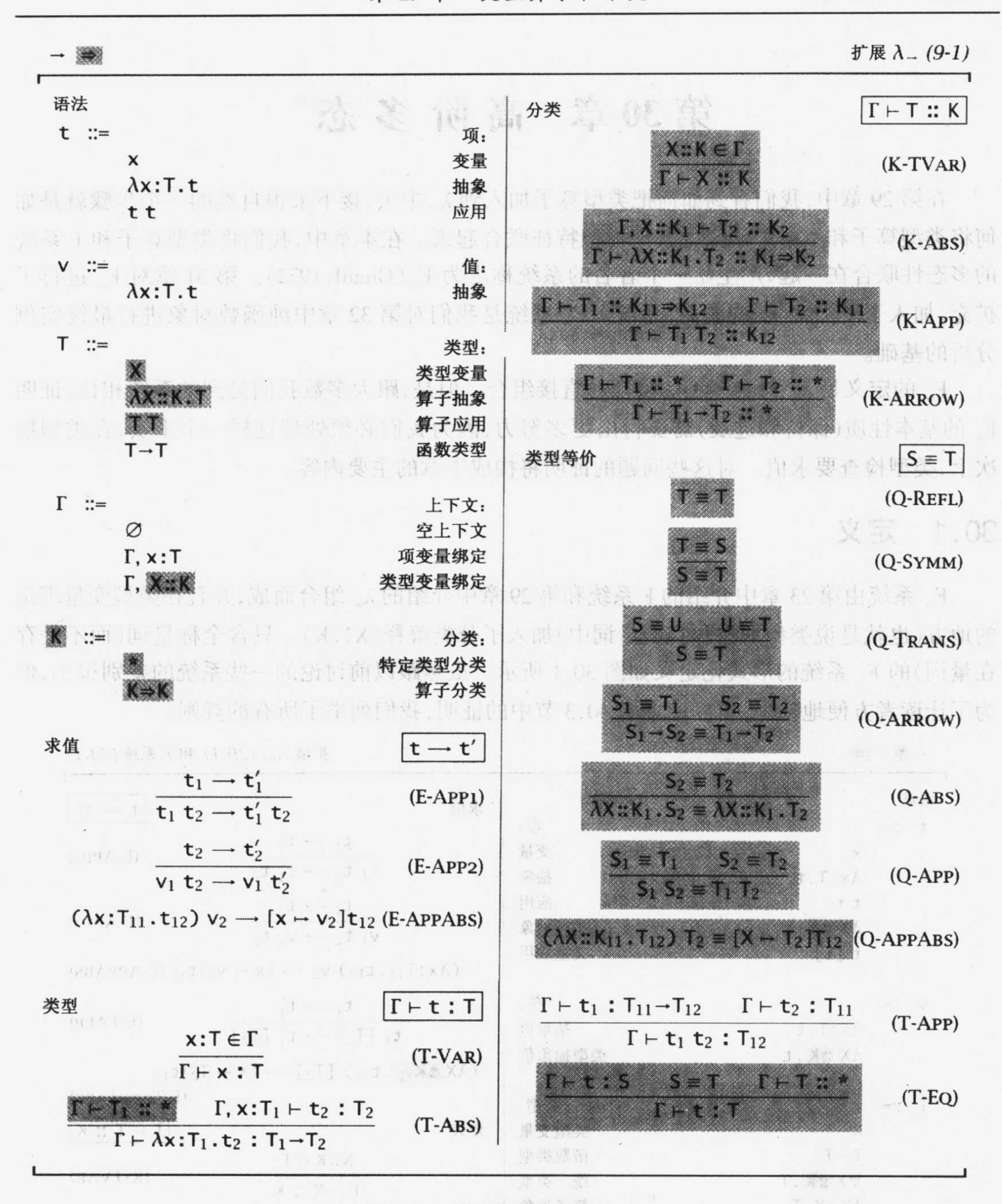


图 29.1　类型算子和分类(λ_ω)

第三,我们增加了一组类型间定义性等价关系的规则。S ≡T 表示“类型 S 和类型 T 在定义上是等价的”。这个关系与项层次上的归约关系相似。新规则 T-Eq 体现出了类型间定义性等价的作用。分类前提(在以前的几节中没有对其讨论)保留了上面提到的不变性,“可类型化的项一定有可分类的类型”。注意这条规则与带子类型化系统中包含规则(T-Sub)的相似点。

对这个系统的基本元理论性质,我们还有一些工作要做,因为类型等价关系在形成项的类型上引入了极大的灵活性。我们把这个讨论推迟到第 30 章来完成。

第 30 章 高阶多态[①]

在第 29 章中，我们看到如何把类型算子加入到 $\lambda_{\to}$ 中去，接下来很自然的一个步骤就是如何将类型算子和本书中研究的其他类型特征联合起来。在本章中，我们将类型算子和 F 系统的多态性联合在一起，产生了一个著名的系统称之为 F_ω(Girard,1972)。第 31 章对 F_ω 进行了扩充，加入了子类型化，构成 $F^{\omega}_{<:}$ 系统。$F^{\omega}_{<:}$ 系统是我们对第 32 章中纯函数对象进行最终实例分析的基础。

F_ω 的定义是 λ_ω 与 F 系统的特征的直接组合。但是，跟大多数我们见到的系统相比，证明 F_ω 的基本性质(保持和进展)需要付出更多努力，因为我们必须处理这样一个事实，在类型层次上，类型检查要求值。对这些问题的证明将构成本章的主要内容。

30.1 定义

F_ω 系统由第 23 章中介绍的 F 系统和第 29 章中介绍的 λ_ω 组合而成，并且在类型变量绑定的地方(也就是说类型抽象和类型量词中)加入了分类解释(X::K)。只含全称量词(而不是存在量词)的 F_ω 系统的形式化定义如图 30.1 所示。虽然跟以前讨论的一些系统的差别很小，但为了让读者方便地参考，更好地理解 30.3 节中的证明，我们列举了所有的规则。

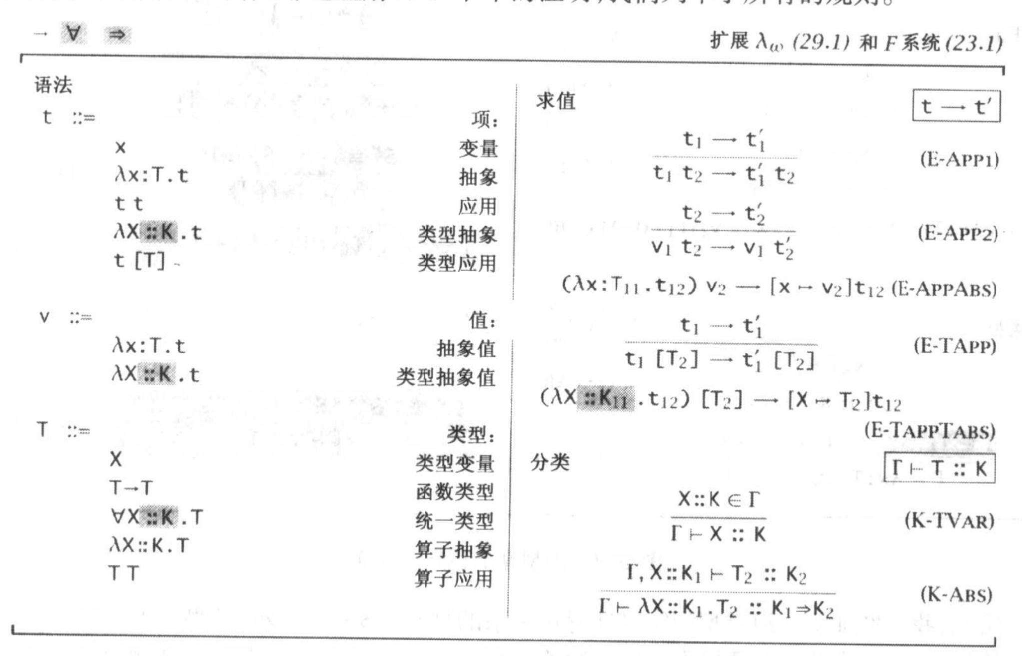


图 30.1 高阶多态 lambda 算法 F_ω

① 本章的例子是根据带记录型、布尔型和存在类型(参见图 30.2)的 F_ω 系统(参见图 30.1)。相关的 OCaml 实现为 fullomega。对 30.5 节提到的依赖类型没有相关的实现。

Γ ::= 上下文:

$\varnothing$ 空上下文

$\Gamma, x{:}T$ 项变量绑定

$\Gamma, X{::}K$ 类型变量绑定

K ::= 分类:

$*$ 特定类型分类

$K \Rightarrow K$ 算子分类

类型等价 $\boxed{S \equiv T}$

$$T \equiv T \quad \text{(Q-REFL)}$$

$$\frac{T \equiv S}{S \equiv T} \quad \text{(Q-SYMM)}$$

$$\frac{S \equiv U \qquad U \equiv T}{S \equiv T} \quad \text{(Q-TRANS)}$$

$$\frac{S_1 \equiv T_1 \qquad S_2 \equiv T_2}{S_1 \to S_2 \equiv T_1 \to T_2} \quad \text{(Q-ARROW)}$$

$$\frac{S_2 \equiv T_2}{\forall X{::}K_1 . S_2 \equiv \forall X{::}K_1 . T_2} \quad \text{(Q-ALL)}$$

$$\frac{S_2 \equiv T_2}{\lambda X{::}K_1 . S_2 \equiv \lambda X{::}K_1 . T_2} \quad \text{(Q-ABS)}$$

$$\frac{S_1 \equiv T_1 \qquad S_2 \equiv T_2}{S_1\ S_2 \equiv T_1\ T_2} \quad \text{(Q-APP)}$$

$$(\lambda X{::}K_{11} . T_{12})\ T_2 \equiv [X \mapsto T_2]T_{12} \quad \text{(Q-APPABS)}$$

$$\frac{\Gamma \vdash T_1 :: K_{11} \Rightarrow K_{12} \qquad \Gamma \vdash T_2 :: K_{11}}{\Gamma \vdash T_1\ T_2 :: K_{12}} \quad \text{(K-APP)}$$

$$\frac{\Gamma \vdash T_1 :: * \qquad \Gamma \vdash T_2 :: *}{\Gamma \vdash T_1 \to T_2 :: *} \quad \text{(K-ARROW)}$$

$$\frac{\Gamma, X{::}K_1 \vdash T_2 :: *}{\Gamma \vdash \forall X{::}K_1 . T_2 :: *} \quad \text{(K-ALL)}$$

continued...

类型化 $\boxed{\Gamma \vdash t : T}$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash T_1 :: * \qquad \Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1 . t_2 : T_1 \to T_2} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{(T-APP)}$$

$$\frac{\Gamma, X{::}K_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda X{::}K_1 . t_2 : \forall X{::}K_1 . T_2} \quad \text{(T-TABS)}$$

$$\frac{\Gamma \vdash t_1 : \forall X{::}K_{11} . T_{12} \qquad \Gamma \vdash T_2 :: K_{11}}{\Gamma \vdash t_1\ [T_2] : [X \mapsto T_2]T_{12}} \quad \text{(T-TAPP)}$$

$$\frac{\Gamma \vdash t : S \qquad S \equiv T \qquad \Gamma \vdash T :: *}{\Gamma \vdash t : T} \quad \text{(T-EQ)}$$

图 30.1 高阶多态 lambda 算法 F_ω(续)

我们把 $\forall X{::}*.T$ 缩写为 $\forall X.T$，$\{\exists X{::}*,T\}$ 记做 $\{\exists X,T\}$，这样 F 系统的项也可以直接读做 F_ω 系统的项。

类似地，我们在存在量词的原始表述中对从 X 到 X::K 的绑定进行归纳，获得了存在量词类型的高阶变式。图 30.2 对这个扩展进行了总结。

30.2 实例

我们将在第 32 章中看到一个编程的扩展实例，编程中用到了涵盖类型算子的抽象。这里先举一个小一点的例子。回忆一下 24.2 节中根据存在量词对抽象数据类型做的编码。假设我们现在想实现一个序对 ADT，其实现方式就与先前实现的计数器类型的 ADT 一样。这个 ADT 应该能够完成构造序对以及拆分序对的操作。另外，这些操作最好是多态的，这样就可以构造和使用任何类型 S 和 T 的元素的序对。也就是说，我们提供的这个抽象类型不应该为合适的类型，而应该是一个抽象类型构造子(或算子)。与先前讨论的 ADT 一样，它也应当是抽象的：对每个 S 和 T，pair 操作把 S 或 T 的元素当成参数，并且返回 Pair S T 的元素。而 fst 和 snd 则把 Pair S T 作为参数相应地返回 S 或 T。抽象成的客户端应当知道它。

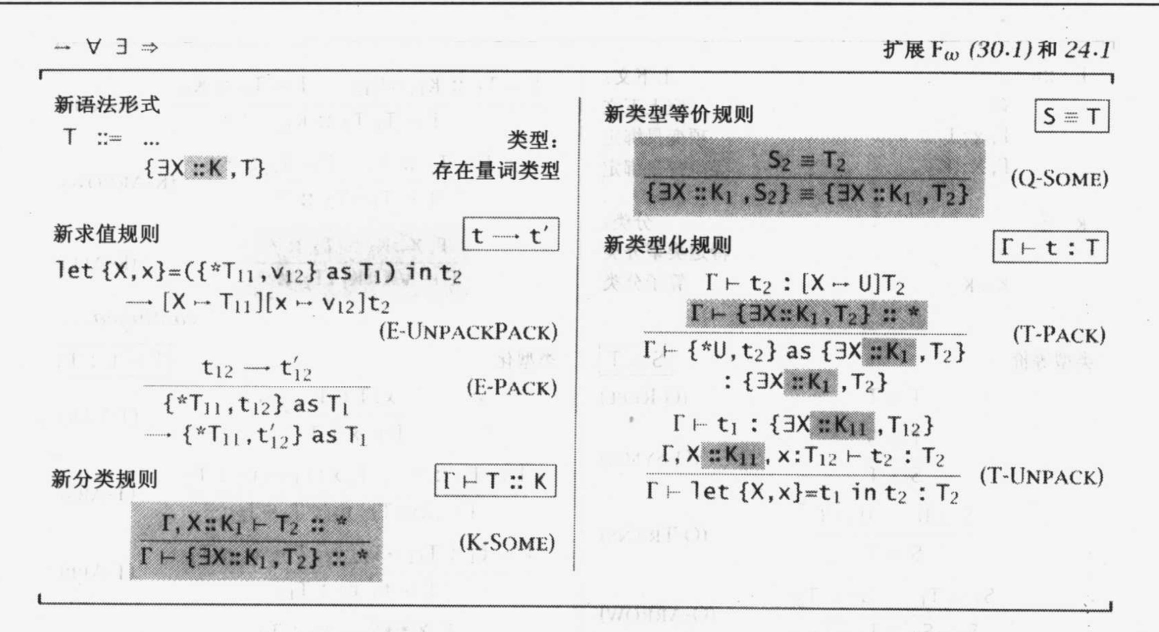


图 30.2 高阶存在量词类型

从这些要求,可以读出我们希望序对 ADT 能表达出的情况:

```
PairSig = {∃Pair::*⇒*⇒*,
           {pair: ∀X. ∀Y. X→Y→(Pair X Y),
            fst: ∀X. ∀Y. (Pair X Y)→X,
            snd: ∀X. ∀Y. (Pair X Y)→Y}};
```

也就是说,一个序对的实现应该提供一个类型算子 Pair 加上给定类型的多态函数 pair,fst 和 snd。

下面是用该类型建立包的方法:

```
pairADT =
    {*λX. λY. ∀R. (X→Y→R) → R,
     {pair = λX. λY. λx:X. λy:Y.
                λR. λp:X→Y→R. p x y,
      fst = λX. λY. λp: ∀R. (X→Y→R) → R.
                p [X] (λx:X. λy:Y. x),
      snd = λX. λY. λp: ∀R. (X→Y→R) → R.
                p [Y] (λx:X. λy:Y. y)}}  as PairSig;
▸ pairADT : PairSig
```

隐藏的表示类型是算子 λX.λY.∀R.(X→Y→R)→R,我们以前(参见 23.4 节)用它来表示过序对。其中 pair,fst 和 snd 是合适的多态函数。

定义完 pairADT,我们可以按通常的方式解开包:

```
let {Pair,pair}=pairADT
in pair.fst [Nat] [Bool] (pair.pair [Nat] [Bool] 5 true);
▸ 5 : Nat
```

30.3 性质

现在来确立 F_ω 系统的基本性质,特别是保持定理和进展定理。这些定理的证明中所包含的思想跟我们以前看到的差不多,但是必须小心进行,因为我们处理的是一个更大更复杂的系

统。特别地，还要花一部分时间去分析类型等价关系的结构。为了缩短证明过程，我们仅处理 F_ω 的全称部分，如图30.1所示。还可直接将证明推广到存在类型。

基本性质

首先来介绍一些简单的性质，随后将用到它们。

30.3.1 引理[加强]：如果 $\Gamma,x:S,\Delta \vdash T::K$，那么 $\Gamma,\Delta \vdash T::K$。

证明：分类关系不涉及项变量绑定。

由于多样性的原因，我们把 F_ω 系统的置换和弱化结合起来证明，而不是像以前那样一个接一个来证明。

30.3.2 引理[置换和弱化]：假定我们有上下文 Γ 和 Δ，其中对某些上下文 Σ 来讲，Δ 是 Γ,Σ 的良形式置换。也就是说 Δ 是 Γ 的一个扩展的置换。

1. 如果 $\Gamma \vdash T::K$，那么 $\Delta \vdash T::K$。
2. 如果 $\Gamma \vdash t:T$，那么 $\Delta \vdash t:T$。

证明：对推导直接进行归纳。

30.3.3 引理[项的代换]：如果 $\Gamma,x:S,\Delta \vdash t:T$，且 $\Gamma \vdash s:S$，那么 $\Gamma,\Delta \vdash [x \mapsto s]t:T$。

证明：对推导直接进行归纳{练习[★]：在何处可用到定理(30.3.1)？那么定理(30.3.2)呢？}

30.3.4 引理[类型代换]：

1. 如果 $\Gamma,Y::J,\Delta \vdash T::K$ 且 $\Gamma \vdash S::J$，那么 $\Gamma,[Y \mapsto S]\Delta \vdash [Y \mapsto S]T::K$。
2. 如果 $T \equiv U$，则 $[Y \mapsto S]T \equiv [Y \mapsto S]U$。
3. 如果 $\Gamma,Y::J,\Delta \vdash t:T$ 且 $\Gamma \vdash S::J$，则 $\Gamma,[Y \mapsto S]\Delta \vdash [Y \mapsto S]t:[Y \mapsto S]T$。

证明：对情况K-TVar和T-Var，使用弱化定理[参见定理(30.3.2)]直接对推导进行归纳。对情况Q-AppAbs，同样可知 $[X \mapsto [Y \mapsto S]T_2]([Y \mapsto S]T_{12})$ 等价于 $[Y \mapsto S]([X \mapsto T_2]T_{12})$。

类型等价和归约

为了确立 F_ω 系统中类型的性质，用一个类型等价关系的有向变式，称为并行归约，十分方便(参见图30.3)。与类型等价的不同在于，去掉了对称规则和传递规则，且规则QR-AppAbs允许约式子语句的归约。去掉对称规则，使归约关系更多"计算的"感觉，$(\lambda X::K_{11}.T_{12})T_2$ 简化成 $[X \mapsto T]T_{12}$；该有向性使得归约关系更容易分析，比如在下面定理(30.3.12)的证明中。去掉传递性，以及同时允许组成部分的归约以简化lambda约式是基于技术上的考虑：我们做出这些变化以得到一步菱形性质的关系，参见定理(30.3.8)。

并行归约关系的一个关键性质是传递闭包和对称闭包，写做($\Leftrightarrow *$)，与类型等价一致。

30.3.5 引理：$S \equiv T$，当且仅当 $S \Leftrightarrow * T$。

证明：$\Leftarrow$方向的证明是显而易见的。对于$\Rightarrow$方向的证明，难点在于一个类型等价推导，也许会在任意点上用到Q-Symm和Q-Trans，而$\Leftrightarrow *$关系的定义中仅允许在最外一层使用对

称和传递。这个问题可如下解决:任意推导 $S \equiv T$ 都可以转换成一系列非传递性推导，$S = S_0 \equiv S_1 \equiv S_2 \equiv \cdots \equiv S_n \equiv T$ 连接在一起利用在上部传递性,其中每个子推导 $S_i \equiv S_{i+1}$，Q-Symm 只用做最后的规则(或根本不用)。

并行归约　　$\boxed{S \Rightarrow T}$

$$T \Rightarrow T \quad \text{(QR-REFL)}$$

$$\frac{S_1 \Rightarrow T_1 \qquad S_2 \Rightarrow T_2}{S_1 \to S_2 \Rightarrow T_1 \to T_2} \quad \text{(QR-ARROW)}$$

$$\frac{S_2 \Rightarrow T_2}{\forall X{::}K_1.S_2 \Rightarrow \forall X{::}K_1.T_2} \quad \text{(QR-ALL)}$$

$$\frac{S_2 \Rightarrow T_2}{\lambda X{::}K_1.S_2 \Rightarrow \lambda X{::}K_1.T_2} \quad \text{(QR-ABS)}$$

$$\frac{S_1 \Rightarrow T_1 \qquad S_2 \Rightarrow T_2}{S_1\ S_2 \Rightarrow T_1\ T_2} \quad \text{(QR-APP)}$$

$$\frac{S_{12} \Rightarrow T_{12} \qquad S_2 \Rightarrow T_2}{(\lambda X{::}K_{11}.S_{12})\ S_2 \Rightarrow [X \mapsto T_2]T_{12}} \quad \text{(QR-APPABS)}$$

图 30.3　类型的并行归约

另外,就如下面的几条引理所示的那样,并行归约可以简单地视为汇合的(汇合通常被称为 Church-Rosser 性质)。

30.3.6　引理: 如果 $S \Rightarrow S'$,那么对任意类型 T 有 $[Y \mapsto S]T \Rightarrow [Y \mapsto S']T$。

证明: 对 T 的结构进行归纳。

30.3.7　引理: 如果 $S \Rightarrow S'$ 且 $T \Rightarrow T'$ 那么 $[Y \mapsto S]T \Rightarrow [Y \mapsto S']T'$。

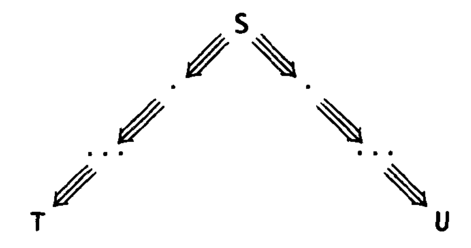
证明: 对给定的第二个推导进行归纳。情况 QR-Refl 用引理(30.3.6)。情况 QR-Abs, QR-App, QR-Arrow, QR-All 直接使用归纳假设进行处理。对情况 QR-AppAbs,有 $T = (\lambda X{::}K_{11}.T_{12})T_2$ 及 $T' = [X \mapsto T_2']T_{12}'$,其中 $T_{12} \Rightarrow T_{12}'$ 且 $T_2 \Rightarrow T_2'$。根据归纳假设,$[Y \mapsto S]T_{12} \Rightarrow [Y \mapsto S']T_{12}'$ 及 $[Y \mapsto S]T_{12} \Rightarrow [Y \mapsto S']T_2'$。应用 QR-AppAbs,有 $(\lambda X{::}K_{11}.[Y \mapsto S]T_{12})[Y \mapsto S]T_2 \Rightarrow [X \mapsto [Y \mapsto S']T_2']([Y \mapsto S']T_{12}')$,即 $[Y \mapsto S]((\lambda X{::}K_{11}.T_{12})T_2) \Rightarrow [Y \mapsto S']([X \mapsto T_2']T_{12}')$。

30.3.8　引理[归约的一步菱形性质]: 如果 $S \Rightarrow T$ 且 $S \Rightarrow U$,那么存在某类型 V 满足 $T \Rightarrow V$ 且 $U \Rightarrow V$。

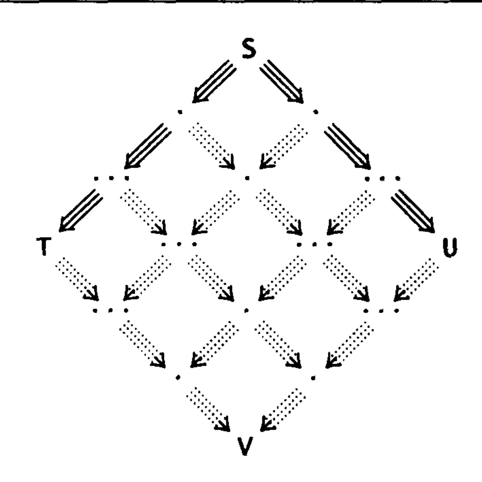
证明: 留做练习[推荐,★★★]。

30.3.9　引理[汇合]: 如果 $S \Rightarrow * T$ 且 $S \Rightarrow * U$,那么存在某类型 V 使 $T \Rightarrow * V$ 和 $U \Rightarrow * V$。

证明: 如果从 S 到 T 以及从 S 到 U 的归约的步骤按如下可视化显示:



那么,我们可以重复使用引理(30.3.8),平铺显示图表的内部,以得到一个大的菱形。该菱形下部的边代表我们需要的归约。



30.3.10 命题:如果 $S \Leftrightarrow^* T$,那么存在某类型 U 使得 $S \Rrightarrow^* U$,且 $T \Rrightarrow^* U$。

证明:留做练习[★★]。

这个命题告诉了我们等价和归约的关系:如果两个类型是等价的,则它们有一个共同的约式。这给了我们一个结构上的提示:还应该证明它的逆转性质。

30.3.11 推论:如果 $S \equiv T$,那么存在某 U 使 $S \Rrightarrow^* U$,$T \Rrightarrow^* U$。

保持

我们现在基本上为证明"在归约中类型是保持的"做好了准备。像往常一样,现在惟一需要的就是逆转引理,对结论具有某个确定形状的推导,逆转引理可以给出其子推导的形状。提出引理之前,先对并行归约做个简单判断。

30.3.12 引理[归约下的形状保持]:

1. 如果 $S_1 \rightarrow S_2 \Rrightarrow^* T$,那么 $T = T_1 \rightarrow T_2$ 其中 $S_1 \Rrightarrow^* T_1$ 且 $S_2 \Rrightarrow^* T_2$。
2. 如果 $\forall X{::}K_1.S_2 \Rrightarrow^* T$,那么 $T = \forall X{::}K_1.T_2$ 其中 $S_2 \Rrightarrow^* T_2$。

证明:直接归纳。

30.3.13 引理[逆转]:

1. 如果 $\Gamma \vdash \lambda x{:}S_1.s_2 : T_1 \rightarrow T_2$,则 $T_1 \equiv S_1$,$\Gamma, x{:}S_1 \vdash s_2 : T_2$ 且 $\Gamma \vdash S_1 :: *$。
2. 如果 $\Gamma \vdash \lambda X{::}J_1.s_2 : \forall X{::}K_1.T_2$,则 $J_1 = K_1$ 且 $\Gamma, X{::}J_1 \vdash s_2 : T_2$。

证明:对于第(1)部分,我们使用归纳来证明。看下面这个更一般的说明:如果 $\Gamma \vdash \lambda x{:}S_1.s_2 : S$ 且 $S \equiv T_1 \rightarrow T_2$,则 $T_1 \equiv S_1$,$\Gamma, x{:}S_1 \vdash s_2 : T_2$。直接使用规则 T-Eq 进行归纳。最有意思的情况是规则 T-Abs,该规则是归纳的基础。在该情况中,S 具有 $S_1 \rightarrow S_2$ 的形式,并且 $\Gamma, x{:}S_1 \vdash s_2 : S_2$。引理[30.3.12(1)]告诉我们 $T_1 \equiv S_1$,$T_2 \equiv S_2$,根据 T-Eq,得到 $\Gamma, x{:}S_1 \vdash s_2 : T_2$。另外,根据 T-Abs 的另外一个前提,我们得到 $\Gamma \vdash S_1 :: *$,由此得证。第(2)部分的证明与此类似。

30.1.14 定理[保持]:如果 $\Gamma \vdash t : T$ 且 $t \rightarrow t'$,则 $\Gamma \vdash t' : T$。

证明:对类型归纳进行直接归纳。证明过程与带子类型化的简单类型 lambda 演算保持的证明相似。

情况 T-VAR: $t = x$

不会发生(没有对变量的求值规则)。

情况 T-ABS: $t = \lambda x{:}T_1.t_2$

不会发生(t 已经是一个值了)。

情况 T-APP: $t = t_1\ t_2$ $\quad \Gamma \vdash t_1 : T_{11}\to T_{12}$ $\quad \Gamma \vdash t_2 : T_{11}$ $\quad T = T_{12}$

从图 30.1 中,我们看到,由规则 E-App1,E-App2,E-AppAbs 能推导出 $t \to t'$。对前两个规则,直接运用归纳假设就可以得到结论。第三个规则更有趣一些:

子情况 E-APPABS: $t_1 = \lambda x{:}S_{11}.\,t_{12}$ $\quad t_2 = v_2$ $\quad t' = [x \mapsto v_2]t_{12}$

根据引理[30.3.13(1)],有 $T_{11} \equiv S_{11}$ 且 $\Gamma, x{:}S_{11} \vdash t_{12} : T_{12}$。由 T-Eq 有 $\Gamma \vdash t_2 : S_{11}$。根据这个结论以及代换引理(30.3.3),得出 $\Gamma \vdash t' : T_{12}$。

情况 T-TABS: $t = \lambda X{::}K_1.t_2$

不会发生(t 已经是一个值了)。

情况 T-TAPP: $t = t_1\ [T_2]$ $\quad \Gamma \vdash t_1 : \forall X{::}K_{11}.T_{12}$ $\quad \Gamma \vdash T_2 :: K_{11}$
$T = [X \mapsto T_2]T_{12}$

与情况 T-App 类似,用类型代换引理(30.3.4)代替项代换引理(30.3.3)

情况 T-EQ: $\Gamma \vdash t : S$ $\quad S \equiv T$ $\quad \Gamma \vdash T :: *$

由归纳假设,$\Gamma \vdash t' : S$。再根据 T-Eq,$\Gamma \vdash t' : T$。

进展

下一个任务是进展定理。同样,我们已经有了证明该定理的大部分内容——缺少的仅有一个典型形式引理,它能给出闭值的形状。

30.3.15 引理[典型形式]

1. 如果 t 是一个带 $\vdash t{:}T_1\to T_2$ 的闭值,则 t 是一个抽象。
2. 如果 t 是一个带 $\vdash t{:}\forall X{::}K_1.T_2$ 的闭值,则 t 是一个类型抽象。

证明:两部分的证明是类似的;我们只证明第(1)部分。因为只有两种形式的值,所以如果 t 是一个值而不是抽象,则 t 肯定是类型抽象。假定(为了推出矛盾)t 是一个类型抽象,那么 $\vdash t{:}T_1\to T_2$ 的给定的类型推导最后必定使用的是规则 T-TAbs 以及一个非空系列的 T-Eq。也就是说,它应该是下面这个形式(忽略分类前提):

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\vdash t : \forall X{::}K_{11}.S_{12}}\ (\text{T-TABS}) \qquad \forall X{::}K_{11}.S_{12} \equiv U_1}{\vdash t : U_1}\ (\text{T-EQ})}{\vdots}}{}$$

$$\dfrac{\dfrac{\vdash t : U_{n-1} \qquad U_{n-1} \equiv U_n}{\vdash t : U_n}\ (\text{T-EQ}) \qquad U_n \equiv T_1\to T_2}{\vdash t : T_1\to T_2}\ (\text{T-EQ})$$

因为类型等价是传递性的，我们可以把所有的等价折合成一个，得到$\forall X::K_{11}.S_{12} \equiv T_1 \to T_2$。现在，由命题(30.3.11)，必然存在某个类型U，使得$\forall X::K_{11}.S_{12} \Rrightarrow^* U$以及$T_1 \to T_2 \Rrightarrow^* U$。再根据引理(30.3.12)，这样的U必须有一个量词和一个箭头作为其最外层的构造子。这是矛盾的。

30.3.16　定理[进展]：假定t是一个封闭的良类型的项(即对某个T，有$\vdash t:T$)，那么t要么是一个值，要么存在某个t'，使得$t \to t'$。

证明：对类型推导进行归纳。情况T-Var不可能发生，因为t是封闭的。情况T-Vbs和T-TAbs直接得到证明，因为抽象就是值。情况T-Eq直接通过归纳假设来证明。剩下的情况只有应用和类型应用，这两种情况更有趣一点。我们给出类型应用的证明，另外一个类似。

情况T-TApp：　$t = t_1\ [T_2]$　　$\vdash t_1 : \forall X::K_{11}.T_{12}$　　$\vdash T_2 :: K_1$

由归纳假设，t_1要么是一个值，要么可以进行一步求值。如果t_1可以进行一步求值，则对t运用规则E-TApp。如果t_1是一个值，则由典型形式引理(30.3.15)，t是一个类型抽象，对t运用E-TAppTAbs。

30.3.17　练习[推荐，★★]：假定我们在类型等价关系中加入下面这个特殊的规则：

$$T \to T \equiv \forall X::*.T$$

系统的哪一条性质将不再成立(如果有的话)？假定加入规则：

$$S \to T \equiv T \to S$$

哪条性质将不再成立(如果有的话)？

分类

在图30.1 F_ω的定义中，我们在保证类型的良分类性方面做了一些努力。使用这些规则对项的推导保证良分类类型。特别地，T-Abs在把对lambda抽象的类型注释加入到上下文之前，对其进行检查，以确定它是不是良形式的。T-Eq检查归属于t的类型T是否具有分类*。保证良形式的准确意义由下列命题给出。

30.3.18　定义：上下文Γ是良形式的，只要(1)Γ是空的，或(2)$\Gamma = \Gamma_1, x:T$，其中Γ_1是良形式的，且$\Gamma \vdash T::*$，或(3)$\Gamma = \Gamma_1, X::K$，其中Γ_1是良形式的。三者满足其一即可。

30.3.19　命题：如果$\Gamma \vdash t:T$且Γ是良形式的，则$\Gamma \vdash T::*$

证明：对T-TApp情况用引理[30.3.4(1)]做常规归纳。

可判定性

由于篇幅限制，本书未包含对F_ω系统可判定性的完整证明，即类型检查算法的可靠性、完备性以及可终止性。但是几乎所有需要的思想都与第28章中介绍的$F_{<:}$系统的最小类型算法相似。

我们注意到分类关系是可判定的(因为其规则是语法制导的)。这没什么好奇怪的，因为我们知道分类其实就是高一层的简单类型lambda演算。这也保证了类型规则中良分类检查可以被有效地实现。

接下来,我们从类型关系中去除一个非语法制导的规则 T-Eq,类似于从 $F_{<:}$ 系统中去掉了规则 T-Sub。然后,将对其余的规则做检查,确定必须对哪些前提做一般化处理,以弥补去掉的 T-Eq 起到的作用。有两处关键点:

1. 在 T-App 和 T-TApp 的第一个前提条件中,我们需要用 T-Eq 重写左边子表达式 t_1 的类型,以把箭头或量词提到外面去[例如,如果上下文把变量 x 与类型(λX.X→X)Nat 联系在一起,则应用 x 5 具有类型 Nat,因为我们可以把 x 的类型重写成 Nat→Nat]。
   通过引入 28.1 节中介绍的揭示关系的一个类似关系来实现重写。这里,不再提升 t_1 的最小类型,直到它成为箭头类型或量词时,我们只对它进行归约——比如重复运用图 30.3 中的规则直到没有非平凡归约的可能性出现[①]。
   为了保证这个化简过程可终止,我们需要证明归约规则是规范化的。当然,对于非良分类的项,归约将不是规范化的,因为 F_ω 系统的类型语法包括所有发散的项,如 omega 进行编码的原语。幸运的是,根据命题(30.3.19),只要我们以良形式的上下文开始(执行合适的分类检查以保证写入上下文中的任意一个注释都是良分类的),只需要处理良分类的项,这些项通常可以归约为惟一的范式(采用第 12 章中的方法)。
2. 在 T-App 的第二个前提条件中,我们可能需要用等价来对 t_2 的类型 T_2 和箭头类型 t_1 中左端 T_{11} 进行匹配。该规则的一个算法变式将对 T_2 和 T_{11} 进行等价检查。等价检查可以如下完成:把 T_2 和 T_{11} 都归约成它们各自的范式,然后测试是不是相同(以囿变量的名字为模)。

30.3.20 练习[★★★★]:基于上述思想,实现 F_ω 系统的类型检查器,以 purefsub 检查器作为出发点。

30.4 F_ω 系统片断

直觉上,$\lambda_\to$ 和 F 系统显然是包含在 F_ω 系统中的。我们在 F_ω 范围内定义一个系统层次结构,F_1,F_2,F_3,来使直觉更加精确。

30.4.1 定义:在系统 F_1 中,惟一的一个分类是 *,类型的量词化(∀)和抽象(λ)都是不允许的。后面的系统根据第 i 层分类来划分层次,定义如下:

$$
\begin{aligned}
\mathcal{K}_1 &= \varnothing \\
\mathcal{K}_{i+1} &= \{*\} \cup \{J \Rightarrow K \mid J \in \mathcal{K}_i \text{ and } K \in \mathcal{K}_{i+1}\} \\
\mathcal{K}_\omega &= \bigcup_{1 \le i} \mathcal{K}_i
\end{aligned}
$$

在系统 F_2 中,仍然只有分类 *,在类型层次上,lambda 抽象仍然是不允许的,但可以对分类 * 的合适类型进行量词化。在系统 F_3 中,允许对类型算子进行量词化(可以写出形如 ∀X::K.T 的类型表达式,其中 $K \in \mathcal{K}_3$),并引入合适类型的抽象(形如 λX::*.T 形式的类型表达式具有分类 *⇒*)。通常,F_{i+1} 系统允许在 $\mathcal{K}_{i+1}$ 分类层次上对带分类的类型进行量化,在 $\mathcal{K}_i$ 分类层次上对分类的类型进行抽象。

① 实际上,F_ω 系统的大部分类型检查使用了归约的保守形式:弱的头归约,即在约式的最左和最外归约,直到某具体构造子(即除了应用外)出现在类型最前端时才停止。

F_1 就是简单类型的 lambda 演算 $\lambda_{\to}$。表面上,它的定义远比图 9.1 中的定义要复杂,因为它包含了分类和类型等价关系,但这些都是微不足道的:每一个含分类 * 语法构成上是良形式的类型也是良分类的,类型 T 只有惟一的等价类型,即它本身。F_2 是我们讨论过的系统 F,因其在分层结构上处的位置,通常称它为二阶 lambda 演算。F_3 系统是第一个分类和等价关系为非退化的系统。

有意思的是,本书的所有程序都是基于 F_3 系统的(严格地讲,第 32 章中介绍类型算子 Object 和 Class 是基于 F_4 系统的,因为其参数是分类(* ⇒ *)⇒ * 的一个类型算子。但我们同样可以以元语言的缩写机制来处理这两个类型算子,而不是以演算的成熟表达式来处理,如在第 29 章对 Pair 的处理,因为在用到 Object 和 Class 的例子中不需要给这种类型量化。另一方面,把编程语言限制在 F_3 系统而不是全 F_ω 系统,并没有对实现难度或元理论复杂度的简化有多大帮助,因为类型算子抽象和类型等价在这层已经存在。

30.4.2　练习[★★★★ ↛]:存不存在一些有用的程序基于 F_4 系统而非 F_3 系统?

30.5　进一步讨论:依赖类型

本书的大部分内容都把精力集中于对各类抽象机制的形式化工作上面了。在简单类型的 lambda 演算中,我们对“取一个项,抽象出子项”操作进行形式化,产生一个函数,随后可以通过把该函数应用到不同的项中,对其进行实例化。在 F 系统中,我们考虑的操作是取一个项,抽象出某个类型,产生一个可被不同类型实例化的项。在 λ_ω 中,我们概括了向上一层简单类型 lambda 的演算机制,取一个类型,抽象出一个子表达式以获取类型算子,通过把类型算子运用到不同的类型中对该算子进行实例化说明。

考虑这些抽象形式的一个方便的方法是表达式簇的概念,由其他的表达式做索引。普通的 lambda 抽象 $\lambda x{:}T_1.t_2$ 就是项$[x \mapsto s]t_1$ 的一个簇,项 s 为索引。类似地,类型抽象 $\lambda X{::}K_1.t_2$ 是由类型做索引的项的簇,类型算子是由类型做索引的类型的簇。

$\lambda x{:}T_1.t_2$	**项做索引的项的簇**
$\lambda X{::}K_1.t_2$	**类型做索引的项的簇**
$\lambda X{::}K_1.T_2$	**类型做索引的类型的簇**

看上面这个例子,显然还有一种刚刚没有讨论过的可能性:项做索引的类型簇。这种抽象形式已经被广泛地研究过了,称为依赖类型。

依赖类型提供了描述程序行为的一个精确度,这跟我们以前见到的其他类型特征完全不同。例如,我们有这样一个内置的类型 FloatList,看如下一些操作:

```
nil    : FloatList
cons   : Float → FloatList → FloatList
hd     : FloatList → Float
tl     : FloatList → FloatList
isnil  : FloatList → Bool
```

在带依赖类型的语言中,可以把简单类型 FloatList 改进成类型 FloatList n 的一个簇(含 n 个元素的列表类型)。

为了利用这个改进，我们加强基本列表的操作类型。首先，赋给常量 nil 类型 FloatList 0。为了给剩下的其他操作更精确的类型，需要改进函数类型的符号，以表达出参数与结果类型之间的依赖性。例如，cons 的类型大概应该为"一个函数取 Float 值和一个长度为 n 的列表为参数，返回长度为 n+1 的列表"。如果我们显式通过给 n 提供一个初始化参数来对 n 进行绑定，则 cons 可以描述成"一函数取数字 n，一个 Float 类型值以及一长度为 n 的队列为参数，返回一个长度为 succ n(n 的后继)的列表。也就是说，在该类型中我们需要得到的是第一个参数(n)的值与第三个参数(FloatList n)和结果[FloatList (succ n)] 的类型之间的依赖关系。我们通过把绑定一个名字到第一个参数上去来实现该操作，记为 Πn:Nat⋯ 而不是 Nat→⋯ 。cons 的类型和其他列表操作则变成：

```
nil  :  FloatList 0
cons :  Πn:Nat. Float → FloatList n → FloatList (succ n)
hd   :  Πn:Nat. FloatList (succ n) → Float
tl   :  Πn:Nat. FloatList (succ n) → FloatList n
```

nil，cons，tl 的类型确切地告诉我们在它们的结果中有多少个元素，hd 和 tl 需要非空列表作为参数。注意，我们不再需要 isnil，因为只要测试一下 n 是否为 0 就可以知道 FloatList n 的一个元素是否为 nil。

$\Pi x{:}T_1.T_2$ 形式的依赖函数类型是箭头类型 $T_1 \to T_2$ 的更精确形式，其中我们绑定一代表函数参数的变量 x，这样就能在结果类型 T_2 中提到 x。在退化的情况中，当 T_2 没有提及 x 时，记 $\Pi x{:}T_1.T_2$ 表示 $T_1 \to T_2$。

当然，我们也可以定义带依赖函数类型的新项。例如，函数：

```
consthree = λn:Nat. λf:Float. λl:FloatList n.
                cons (succ(succ n)) f
                  (cons (succ n) f
                     (cons n f l));
```

在其第三个参数(f)前预先考虑了第二个参数(f)的三份拷贝，类型为：

```
Πn:Nat. Float → FloatList n → FloatList (succ(succ(succ n)))
```

注意，三次调用 cons 时的第一个参数均不相同，反映了这三次调用的列表参数长度也都不相同。在计算机科学与逻辑学上，存在关于依赖类型的广泛文献。有一些值得借鉴，如 Smith，Nordström 和 Petersson(1990)，Thompson(1991)，Luo(1994)和 Hofmann(1997)。

30.5.1 练习[★★]：把列表元素的类型固定成 Float 类型可以使例子简单，但是我们用类型算子把它一般化成任意类型 T 的列表。请说明该如何做。

按照同样的步骤，我们可以使用类似的改进类型来构造更高层的列表操作函数。例如，能构造一分类函数，其类型为：

```
sort : Πn:Nat. FloatList n → FloatList n,
```

该类型告诉我们函数将返回跟输入长度相同的列表。事实上，通过对包含的类型簇的进一步改进，甚至可以构造出 sort 函数，其类型列表明该函数返回的队列通常是经过排序的。sort 函数属于该类型，也就意味着是证明该函数满足说明。

这样的例子描述了一个诱人的画面:程序按结构来讲都是正确的,其中程序的类型告诉了大家想知道的程序行为,类型检查器返回OK,使大家有信心认为程序运行正常。还说明这样的一种编程思想:在证明"说明的可满足性"时,把计算的内容提取出来。形为"对任意一个 x 存在一个 y,使 P"的构造性定理证明可以被视为函数:把 x 映射到 y,并且有证据(在计算上无关紧要,仅有类型检查器对它感兴趣)表明该函数有性质 P。这些思想已经由Nuprl(Constable等,1986),LEGO(Luo和Pollack,1992,Pollack,1994)以及Cop(Paulin-Mohring,1989)项目的研究人员研究过了。

遗憾的是,依赖类型带来的是双重麻烦。类型检查与定理证明之间的区别被混淆后并没使证明简化。相反地,它使得类型检查在计算上难以驾御。使用机器证明作为助手的数学家不仅仅是输入一条定理,按个按钮,坐回到椅子上等待Yes或No:他们需要付出相当大的努力写出证明手稿和策略来构造和核实证明。如果我们也按这种想法,程序员应该付出同样多的努力对程序进行注释,给出解释和提示,以引导类型检查器。对某些关键的编程任务,这样付出努力是合适的,但对于日常编程来讲,这个代价显得太昂贵了。

无论怎样,在实际编程语言的设计中,依赖类型已有一些应用,包括Russell(Donahue和Demer,1985; Hook,1984),Cayenne(Augustsson,1998),Dependent ML(Xi和Pfenning,1998,1999),依赖类型化组合语言(Xi和Harper,2001),以及Jay和Sekanina(1997)的shape类型。这些语言以多种方式限制依赖类型,得到更易处理的系统,其类型检查可以更好地自动执行。例如,在Xi等语言中,依赖类型仅用在检查数组存取运行时间范围的静态消除上。在这些语言中,类型检查时出现的定理证明问题是一组线性约束,对这些约束存在好的自动过程。

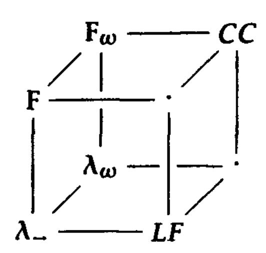
依赖类型对编程语言有长时间影响的一个领域是在模块系统的设计过程中,该模块系统包含了追踪内部模块依赖的体制。在该领域的里程碑包括Pebble(Burstall和Lampson,1984),MacQueen(1986),Mitchell和Harper(1988),Harper等(1990)以及Harper和Stone。该领域的最近的一些论文已经采用了单一分类的技术手段,模块依赖在分类层次而不是类型层次上实现(例如,Stone和Harper,2000;Crary,2000;在Hayashi,1991,Aspinall,1994也可以看到)。

依赖类型和子类型化的组合方式由Cardelli(1998b)首先提出,而Aspinall(1994),Pfenning(1993b),Aspinall和Compagnoni(2001),Chen和Longo(1996)以及Zwanenburg(1999)对其做了进一步研究和一般化处理。

计算机科学中依赖类型的另外一个重要的应用是建立证明助手和定理自动证明机。特别地,带依赖类型的简单类型系统通常被称为逻辑框架。最著名的是带依赖类型的纯简单类型演算LF(Harper,Honsell和Plotkin,1992)。LF及其相关的性质,特别是构造演算(Coquand和Huet,1988;Luo,1994),构成了定理证明环境的基础,包含AutoMath(de Bruijn,1980),NuPRL(Constable等,1986),LEGO(Luo和Pollack,1992; Pollack,1994),Coq(Barras等,1997),ALF(Magnusson和Nordström,1994),以及ELF(Pfenning,1994)。Pfenning(1996)在该领域做了更深入的综述。

在本章前面讨论的四种抽象形式可以由下图表示,称为Barendregt立方体[①]:

① Barendregt(1991)称之为λ立方体。



立方体中所有的系统都包含普通的项抽象。上表面代表带多态性的系统(由类型做索引的项簇),下表面代表带类型算子的系统,右表面代表带依赖类型的系统。远端右边的角代表构造演算,包含所有的四种抽象形式。在上面提到的另一个角是 LF,即简单类型的 lambda 演算。Barendregt 立方体表示的所有系统,以及其他一些系统,可以表示为纯类型系统的一般框架的实例(Terlouw,1989;Berardi,1988;Barendregt 1991,1992; Jutting,McKinna 和 Pollack,1994; McKinna 和 Pollack,1993; Pollack,1994)。

第 31 章　高阶子类型化[①]

我们需要思考的最后一个系统叫做 $F^{\omega}_{<:}$(F-omega-sub),仍然是我们以前独立研究过的一些特征的组合,这些特征是类型操作子和子类型的组合形式。有了类型操作子,这种组合可视为 $F_{<:}$ 系统的扩展,是含有囿量词的二阶多态的 lambda 演算。最有趣的新特征是从分类 * 到更高分类的类型子类型化关系的扩充。

目前出现了多种不同版本的 $F^{\omega}_{<:}$ 系统,它们在表达能力和元理论复杂度上有差异,在这里使用非常接近于 Pierce 和 Steffen(1994)提出的版本,也是最简单的版本之一。我们将不证明该系统的任何性质;有兴趣的读者可以参考 Pierce 和 Steffen(1994),或论述类似系统的 Compagnoni(1994)或 Abadi 和 Cardelli(1996)(如果将 30.3 节的证明复杂度乘上第 28 章的复杂度,可以想像出该系统的证明需要占多大的空间)。

讨论 $F^{\omega}_{<:}$ 系统的主要原因是,它形成了在面向对象编程中最新实例学习的框架(参见第 32 章)。所有的例子都不涉及到 $F^{\omega}_{<:}$ 系统定义中的较深部分——所需要的只是在给定的类型操作子的子类型范围内写出囿量词的能力。因此,读者可以在第一次阅读时跳过这一章,以后有问题时再回来阅读。

31.1　直觉

子类型化和带类型操作子的囿量词组合之后,系统在形式化设计方面出现了新的问题。在介绍系统的定义之前我们简要地讨论这些问题。

第一个问题是:有了子类型,像 $\lambda X::K_1.T_2$ 这样的类型操作子是否应被概括化为 $\lambda X<:T_1.T_2$ 的囿类型操作子。在本章中我们只用简单方式(而不用正规方式)定义一个具有囿量词和非囿类型操作子的系统。

下一个问题是如何加入类型操作子来扩展子类型关系。这里有几种选择,我们使用的最简单的一种是在合适类型上逐点将子类型关系提升到类型关系操作子。对 $\lambda X.S$ 和 $\lambda X.T$,如果将它们应用于参数 U,产生的类型为子类型关系,则认为 $\lambda X.S$ 为 $\lambda X.T$ 的子类型。例如,对于任意的 U,因为 Top→U 是 U→Top 的子类型,所以 $\lambda X.\text{Top}\to X$ 是 $\lambda X.X\to\text{Top}$ 的子类型。同样,X 抽象不做任何关于它的子类型或超类型的假设,如果 S 是 T 的子类型,那么可以认为 $\lambda X.S$ 是 $\lambda X.T$ 的子类型。后一种观点直接引出下面的规则:

$$\frac{\Gamma, X \vdash S <: T}{\Gamma \vdash \lambda X.S <: \lambda X.T} \qquad \text{(S-Abs)}$$

相反,如果 F 和 G 是类型操作子且 F <: G,那么 FU <: GU。

$$\frac{\Gamma \vdash F <: G}{\Gamma \vdash F\,U <: G\,U} \qquad \text{(S-App)}$$

① 本章中研究的系统是纯 $F^{\omega}_{<:}$(参见图 31.1)。相应的实现是 fomsub(fullfomsub 实现包含多种扩展,如存在量词)。

注意,只有 F 和 G 被应用于同样的参数 U 时,这条规则才生效;当参数不同时,即使 F 是 G 的逐点子类型,也不能说明满足该规则(31.4 节中提出的一些更复杂的 $F^{\omega}_{<:}$ 变式就考虑了这种情况)。

类型等价关系还会产生一条附加规则。如果 S ≡ T,那么 S 和 T 有相同的成员,但有相同成员的类型一定互为子类型。这就引出另一条子类型化规则,它将定义性等价关系作为基本情况包含在内:

$$\frac{\Gamma \vdash S :: K \quad \Gamma \vdash T :: K \quad S \equiv T}{\Gamma \vdash S <: T} \qquad \text{(S-Eq)}$$

完成了从分类 * 到分类 * ⇒ * 的子类型化提升,就可以对更复杂的分类重复这一过程。例如:如果 P 和 Q 是分类 * ⇒ * ⇒ * 中的类型操作子,那么,如果对任意的 U,应用 PU 在分类 * ⇒ * 中是 QU 的子类型,我们就认为有 P <: Q。

该定义有力地说明了更高分类的子类型关系都有最大的元素。如果设 Top[*] = Top,而且定义(高分类的最大元素):

$$\text{Top}[K_1 \Rightarrow K_2] \stackrel{\text{def}}{=} \lambda X :: K_1 . \text{Top}[K_2],$$

那么一个简单的归纳可说明 $\Gamma \vdash S <: \text{Top}[K]$(当 S 具有分类 K 时)。下一节中,我们将把该效果加入到规则中。

从普通的囿量词到高阶囿量词可直接转变。$F^{\omega}_{<:}$ 系统从 $F_{<:}$ 系统中继承了形式为 $\forall X <: T_1 . T_2$ 的囿量词。概括化到高阶(即类型操作子的量词)对这一语法不做任何修改:我们刚观察到这里的 T_1 可以是任意的类型表达方式,包括一个类型操作子。从 F_{ω} 继承来的非囿高阶量词可以被当做具有最大边界的囿量词的缩写,即把 $\forall X :: K_1 . T_2$ 当做 $\forall X <: \text{Top}[K_1] . T_2$ 的缩写。

最后,$F^{\omega}_{<:}$ 系统出现了与 $F_{<:}$ 系统相同的问题,即使用 S-All 规则更容易处理核心变式,还是用功能更强大的全变式。在这里我们选择用核心变式;全变式虽然在语义上是合理的,但它的元理论性质(甚至那些与全 $F_{<:}$ 系统相比之下也应该具备的性质)仍然没有建立。

31.2 定义

图 31.1 列出了定义 $F^{\omega}_{<:}$ 的规则。该定义有一个技术问题:虽然系统提出了类型变量的两种不同的绑定(类型操作子中的 X::K 和囿量词中的 X <: T),但我们在上下文中只采用后一种绑定。当我们将一个 X::K 绑定器从 ↦ 的右端移动到左端时,根据规则 K-Abs 和 S-Abs,将它变为 X <: Top[K]。

另一个有意义的地方是 $F_{<:}$ 系统中的 S-Refl 规则和 F_{ω} 系统中的 T-Eq 规则在 $F^{\omega}_{<:}$ 系统中被忽略。老的 S-Refl 规则实例可以从 S-Eq 和 Q-Refl 中直接得出,同时 T-Eq 规则也可以通过 T-Sub 和 S-Eq 推导出。

31.2.1 练习[★]:如果我们定义 Id = λX.X,且:

Γ = B<:Top, A<:B, F <: Id

那么下面哪些子类型描述是可导出的?

$$\begin{array}{lcl}
\Gamma \vdash A & <: & Id\ B \\
\Gamma \vdash Id\ A & <: & B \\
\Gamma \vdash \lambda X.X & <: & \lambda X.Top \\
\Gamma \vdash \lambda X.\ \forall Y<:X.\ Y & <: & \lambda X.\ \forall Y<:Top.\ Y \\
\Gamma \vdash \lambda X.\ \forall Y<:X.\ Y & <: & \lambda X.\ \forall Y<:X.\ X \\
\Gamma \vdash F\ B & <: & B \\
\Gamma \vdash B & <: & F\ B \\
\Gamma \vdash F\ B & <: & F\ B \\
\Gamma \vdash \forall F<:(\lambda Y.Top\to Y).\ F\ A & <: & \forall F<:(\lambda Y.Top\to Y).\ Top\to B \\
\Gamma \vdash \forall F<:(\lambda Y.Top\to Y).\ F\ A & <: & \forall F<:(\lambda Y.Top\to Y).\ F\ B \\
\Gamma \vdash Top[*\Rightarrow *\Rightarrow *] & <: & Top[*\Rightarrow *\Rightarrow *]
\end{array}$$

→ ∀ ⇒ <: Top　　　　基于 F_ω *(30.1)* 和核心 $F_{<:}$ *(26.1)*

语法

t ::= 　　　项：
- x　　变量
- λx:T.t　　抽象
- t t　　应用
- λX<:T.t　　类型抽象
- t [T]　　类型应用

v ::= 　　　值：
- λx:T.t　　抽象值
- λX<:T.t　　类型抽象值

T ::= 　　　类型：
- Top　　最大类型
- X　　类型变量
- T→T　　函数类型
- ∀X<:T.T　　全称类型
- λX::K.T　　操作子抽象
- T T　　操作子应用

Γ ::= 　　　上下文：
- ∅　　空上下文
- Γ, x:T　　项变量绑定
- Γ, X<:T　　类型变量绑定

K ::= 　　　分类：
- *　　适当类型分类
- K⇒K　　操作子分类

类型等价　　$S \equiv T$

$$T \equiv T \quad \text{(Q-REFL)}$$

$$\frac{T \equiv S}{S \equiv T} \quad \text{(Q-SYMM)}$$

求值　　$t \longrightarrow t'$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-APP1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(E-APP2)}$$

$$(\lambda x{:}T_{11}.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \quad \text{(E-APPABS)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ [T_2] \longrightarrow t_1'\ [T_2]} \quad \text{(E-TAPP)}$$

$$(\lambda X{<:}T_{11}.t_{12})\ [T_2] \longrightarrow [X \mapsto T_2]t_{12} \quad \text{(E-TAPPTABS)}$$

分类化　　$\Gamma \vdash T :: K$

$$\Gamma \vdash Top :: * \quad \text{(K-TOP)}$$

$$\frac{X{<:}T \in \Gamma \qquad \Gamma \vdash T :: K}{\Gamma \vdash X :: K} \quad \text{(K-TVAR)}$$

$$\frac{\Gamma, X{<:}Top[K_1] \vdash T_2 :: K_2}{\Gamma \vdash \lambda X{::}K_1.T_2 :: K_1 \Rightarrow K_2} \quad \text{(K-ABS)}$$

$$\frac{\Gamma \vdash T_1 :: K_{11}\Rightarrow K_{12} \qquad \Gamma \vdash T_2 :: K_{11}}{\Gamma \vdash T_1\ T_2 :: K_{12}} \quad \text{(K-APP)}$$

$$\frac{\Gamma \vdash T_1 :: * \qquad \Gamma \vdash T_2 :: *}{\Gamma \vdash T_1 \to T_2 :: *} \quad \text{(K-ARROW)}$$

$$\frac{\Gamma, X{<:}T_1 \vdash T_2 :: *}{\Gamma \vdash \forall X{<:}T_1.T_2 :: *} \quad \text{(K-ALL)}$$

continued...

$$\frac{\Gamma \vdash U_1 :: K_1 \qquad \Gamma, X{<:}U_1 \vdash S_2 <: T_2}{\Gamma \vdash \forall X{<:}U_1.S_2 <: \forall X{<:}U_1.T_2} \quad \text{(S-ALL)}$$

$$\frac{\Gamma, X{<:}Top[K_1] \vdash S_2 <: T_2}{\Gamma \vdash \lambda X{::}K_1.S_2 <: \lambda X{::}K_1.T_2} \quad \text{(S-ABS)}$$

图 31.1　高阶囿量词($F^{\omega}_{<:}$)

$$\frac{S \equiv U \quad U \equiv T}{S \equiv T} \quad \text{(Q-TRANS)}$$

$$\frac{S_1 \equiv T_1 \quad S_2 \equiv T_2}{S_1 \to S_2 \equiv T_1 \to T_2} \quad \text{(Q-ARROW)}$$

$$\frac{S_1 \equiv T_1 \quad S_2 \equiv T_2}{\forall X<:S_1.S_2 \equiv \forall X<:T_1.T_2} \quad \text{(Q-ALL)}$$

$$\frac{S_2 \equiv T_2}{\lambda X::K_1.S_2 \equiv \lambda X::K_1.T_2} \quad \text{(Q-ABS)}$$

$$\frac{S_1 \equiv T_1 \quad S_2 \equiv T_2}{S_1\ S_2 \equiv T_1\ T_2} \quad \text{(Q-APP)}$$

$$(\lambda X::K_{11}.T_{12})\ T_2 \equiv [X \mapsto T_2]T_{12} \quad \text{(Q-APPABS)}$$

子类型化 $\boxed{\Gamma \vdash S <: T}$

$$\frac{\Gamma \vdash S <: U \quad \Gamma \vdash U <: T \quad \Gamma \vdash U :: K}{\Gamma \vdash S <: T} \quad \text{(S-TRANS)}$$

$$\frac{\Gamma \vdash S :: *}{\Gamma \vdash S <: Top} \quad \text{(S-TOP)}$$

$$\frac{\Gamma \vdash T_1 <: S_1 \quad \Gamma \vdash S_2 <: T_2}{\Gamma \vdash S_1 \to S_2 <: T_1 \to T_2} \quad \text{(S-ARROW)}$$

$$\frac{X<:T \in \Gamma}{\Gamma \vdash X <: T} \quad \text{(S-TVAR)}$$

$$\frac{\Gamma \vdash S_1 <: T_1}{\Gamma \vdash S_1\ U <: T_1\ U} \quad \text{(S-APP)}$$

$$\frac{\Gamma \vdash S :: K \quad \Gamma \vdash T :: K \quad S \equiv T}{\Gamma \vdash S <: T} \quad \text{(S-EQ)}$$

类型化 $\boxed{\Gamma \vdash t : T}$

$$\frac{x:T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash T_1 :: * \quad \Gamma, x:T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x:T_1.t_2 : T_1 \to T_2} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{(T-APP)}$$

$$\frac{\Gamma, X<:T \vdash t_2 : T_2}{\Gamma \vdash \lambda X<:T.t_2 : \forall X<:T.T_2} \quad \text{(T-TABS)}$$

$$\frac{\Gamma \vdash t_1 : \forall X<:T_{11}.T_{12} \quad \Gamma \vdash T_2 <: T_{11}}{\Gamma \vdash t_1\ [T_2] : [X \mapsto T_2]T_{12}} \quad \text{(T-TAPP)}$$

$$\frac{\Gamma \vdash t : S \quad \Gamma \vdash S <: T \quad \Gamma \vdash T :: *}{\Gamma \vdash t : T} \quad \text{(T-SUB)}$$

图 31.1 高阶囿量词($F^{\omega}_{<:}$)(续)

31.3 性质

$F^{\omega}_{<:}$ 基本性质的证明,包括系统相关的归约、进展和最小类型化过程中类型保持性质的证明,都可以在本章开始引用的文章中找到。当然这些证明必须分别解决子类型化、囿量词和类型操作子引发的所有问题。另外,在我们试图定义一种可供选择的,语法制导的子类型化规则的表现形式时,发现了一个新的复杂问题:那就是,不仅类型变量规则能够在子类型化推导中以一种必需的方式,与传递性结合使用(如在 28.3 节中看到的),而且类型的等价(参见规则 S-Eq)也可以与传递性结合。

例如:在上下文 $\Gamma = X <: Top, F <: \lambda Y.Y$ 中,语句 $\Gamma \vdash FX <: X$ 可证明如下(忽略分类):

$$\cfrac{\cfrac{\cfrac{}{\Gamma \vdash F <: \lambda Y.Y}\ \text{S-TVAR}}{\Gamma \vdash F\,X <: (\lambda Y.Y)\ X}\ \text{S-APP} \qquad \cfrac{}{\Gamma \vdash (\lambda Y.Y)\ X <: X}\ \text{S-EQ}}{\Gamma \vdash F\,X <: X}\ \text{S-TRANS}$$

而且,我们不能简单地将所有类型表达式归约为范式,因为表达式 F A 不是约式,只有在类型检查过程中,变量 F 被提升到它的上界 λY.Y 时才是约式。解决的方法是在子类型检查开始时就对类型表达式进行一次规范化,并且在提升操作中如果需要的话再进行一次规范化。

31.4 注释

许多 $F^{\omega}_{<:}$ 中的思想都来源于 Cardelli，特别是他的文章"Structural Subtyping and the Notion of Power Type"(构造的子类型化和功能强大的类型的构想，1998a)；子类型关系到类型操作子的扩展是由 Cardelli(1999)和 Mitchell(1999a)研究而来的。Cardelli 和 Longo(1991)利用部分等价关系提出了一个早期的语义模型，Compagnoni 和 Pierce(1996)提出用交叉类型扩展 $F^{\omega}_{<:}$ 的模型。一种更为强大的包括了递归类型的模型由 Bruce 和 Mitchell(1992)提出；在 Abadi 和 Cardelli (1996)的著作中也能找到相关的模型。

这里给出的 $F^{\omega}_{<:}$ 变式的基本元理论性质都曾被 Pierce 和 Steffen(1994)证明过。而且 Compagnoni(1994)也独立证明过(使用一种更为聪明的证明技术，即简化那些主要参数中的一个)。这种技术也被 Abadi 和 Cardelli(1996)使用过，他们证明的 $F^{\omega}_{<:}$ 变式用对象演算而非 lambda 演算作为核心项语言。

如果我们改进分类系统使它可以记录类型操作子的极性，那么类型操作子之间的子类型化的逐点定义就可被概化为不同类型操作子，应用到不同参数(FS <: GT)的应用之间的子类型化。我们认为当 S <: T，如果有 FS <: FT 则 F 是协变式，而如果 FT <: FS，则 F 是逆变式。如果我们引入反映这些性质的两个新的子类型化规则：

$$\frac{\Gamma \vdash S <: T \qquad F\text{ 是协变式}}{\Gamma \vdash F\,S <: F\,T}$$

$$\frac{\Gamma \vdash S <: T \qquad F\text{ 是逆变式}}{\Gamma \vdash F\,T <: F\,S}$$

那么(根据传递性)产生的结果为：如果 F <: G，S <: T，那么 FS <: GT 且 G 是协变式。为了完成这些工作，我们需要用它们的极性来标记类型变量，同时也要将高阶量词限制在具有特定极性的操作子的范围内。有极性的 $F^{\omega}_{<:}$ 的版本在 Cardelli(1990)，Steffen(1998)和 Duggan 和 Compagnoni(1999)中提到。

这里用到的另一种 $F^{\omega}_{<:}$ 的概化描述是将非囿的类型操作子 $\lambda X::K_1.T_2$ 概化为囿的类型操作子 $\lambda X<:T_1.T_2$。这是一个吸引人的步骤，因为这样做正好符合前面在 F 系统中增加子类型化来形成 $F_{<:}$ 的同时，将量词概化为囿量词时的做法。另一方面，因为我们必须概括分类系统使之包括形如 $\forall X<:T_1.K_2$ 的分类，所以它实质上使系统复杂化了；这样就引起了分类化和子类型规则之间的相互依赖，这将带来大量的问题需要解决。参见 Compagnoni 和 Goguen(1997a；1997b)。

Chen 和 Longo(1996)还有 Zwanenburg(1999)研究过含依赖类型的 $F^{\omega}_{<:}$ 扩展。

第 32 章 实例学习:纯函数对象[①]

最后一章实例学习将继续讨论存在对象模型。这个模型已在 24.2 节中简要介绍过了,它说明了如何将存在包视为简单对象,并且将该抽象风格的性质与存在量词的用法之间做对比,以实现更传统的抽象数据类型。在本章中,我们使用前几章中开发出来的一些工具(类型操作子和高阶子类型,加上在 32.7 节中介绍的一种新的特点,多态更新)来将这些简单的存在对象扩展到一个概念集合,使其包含类和继承,从而具备成熟的面向对象编程的灵活性。

32.1 简单对象

我们首先回忆在 24.2 节中的纯函数对象 Counter 的类型:

```
Counter = {∃X, {state:X, methods:{get:X→Nat,inc:X→X}}};
```

这个类型的元素是一些包,这些包拥有一个隐藏的状态类型 X,一个类型 X 的状态和一个类型为{get:X→Nat,inc:X→X}的方法记录。

本章开始几节将把{x:Nat}作为所有对象的表示类型(在 32.8 节中,将看到如何定义带多实例变量的对象,以及增加新的实例变量的类)。当我们讨论内部状态的类型时,将继续使用缩写形式 CounterR。

```
CounterR = {x:Nat};
```

一个计数器对象为 Counter 类型的一个元素,它的定义依据存在性规则(参见图 24.1 的 T-Pack 规则)。

```
c = {*CounterR,
     {state = {x=5},
      methods = {get = λr:CounterR. r.x,
                 inc = λr:CounterR. {x=succ(r.x)}}}} as Counter;
▸ c : Counter
```

调用一个 Counter 的方法包括对它进行解包,从它的方法中选择合适的字段,然后将其应用到状态中:

```
sendget = λc:Counter.
            let {X,body} = c in
            body.methods.get(body.state);
▸ sendget : Counter → Nat
```

最后(对于 inc,它必须返回一个新的对象,而不仅仅是一个数值)将结果按照与原对象相同的表示类型和方法打包到一个新的对象中。

---

① 本章中的例子是有记录、数字和多态更新(参见图 32.1)的 $F^{\omega}_{<:}$ 的项。相关的 OCaml 实现是 fullupdate。

```
sendinc = λc:Counter.
             let {X,body} = c in
               {*X,
                {state = body.methods.inc(body.state),
                 methods = body.methods}} as Counter;
▸ sendinc : Counter → Counter
```

这些基本函数可用来构造操作 Counter 对象的更复杂的项：

```
addthree = λc:Counter. sendinc (sendinc (sendinc c));
▸ addthree : Counter → Counter
```

32.2 子类型化

对象有了存在性编码就具有一个好的特征：由于存在类型及记录类型的子类型化规则直接得出的对象类型之间存在着子类型包含关系。为了检查这一特点，让我们回忆一下（参见图 26.3）存在类型的子类型化规则[①]：

$$\frac{\Gamma, X<:U \vdash S_2 <: T_2}{\Gamma \vdash \{\exists X<:U, S_2\} <: \{\exists X<:U, T_2\}} \qquad \text{(S-Some)}$$

这一规则立刻告诉我们，如果定义一个比 Counter 还要多的方法的对象类型，比如：

```
ResetCounter =
  {∃X, {state:X, methods:{get: X→Nat, inc:X→X, reset:X→X}}};
```

那么它将是 Counter 的子类型，即 ResetCounter <: Counter 。这意味着，如果我们定义一个重置计数器对象：

```
rc = {*CounterR,
      {state = {x=0},
       methods = {get = λr:CounterR. r.x,
                  inc = λr:CounterR. {x=succ(r.x)},
                  reset = λr:CounterR. {x=0}}}} as ResetCounter;

▸ rc : ResetCounter
```

我们可以使用包含将这个对象传递给 Counter 上定义的函数，诸如：sendget，sendinc，addthree：

```
rc3 = addthree rc;

▸ rc3 : Counter
```

注意，尽管如此，当这样做的时候，我们丢失了一些类型信息：这里 rc3 的类型是 Counter，而不是 ResetCounter。

32.3 囿量词

当然，很明显，这一类信息的丢失主要是由于包含，因为在第 26 章中，包含导致了囿量词的出现。尽管如此，仅依靠囿量词是远远不够的，为更有效地解决眼前的问题还需要增加一些新的机制。

---

① 我们这里只用规则的核心变式；暂时用不上全变式。事实上，在本章中，我们一点也不需要囿存在量词——所有的存在量词边界都是 Top。

原因是使用了囿量词的 sendinc 类型的明显修改结果为 $\forall$ C <: Counter. C→C。如果我们有一个这样类型的 sendinc，可以将 addthree 写成：

```
addthree = λC<:Counter. λc:C.
                sendinc [C] (sendinc [C] (sendinc [C] c));

▸ addthree : ∀C<:Counter. C → C
```

并且将它们应用到 rc 中来获得类型为 ResetCounter 的结果：

```
rc3 = addthree [ResetCounter] rc;

▸ rc3 : ResetCounter
```

遗憾的是，我们并没有办法写出这样一个函数，或者说，没有办法写出一个函数能够完成我们希望的功能且能将此类型传递给这个函数。当然，可以写一个属于此类型的恒等函数：

```
wrongsendinc = λC<:Counter. λc:C. c;

▸ wrongsendinc : ∀C<:Counter. C → C
```

但如果要通过在 sendinc 前端增加一个囿类型抽象来改进 sendinc 的实际实现，类型检查时会报错误：

```
sendinc =
   λC<:Counter. λc:C.
     let {X,body} = c in
       {*X,
        {state = body.methods.inc(body.state),
         methods = body.methods}}
       as C;

▸ Error: existential type expected
```

问题出在最后一行，注释 as C 告诉类型检查器“此处创建的包使用存在的类型 C”。但 C 并不是一个存在类型——它只是一个类型变量。这个愚蠢的限制不是已定义的类型规则造成的，例如，规则不“知道”每一个存在类型的子类型都是一个存在类型。相反，如果将类型 C 给包：

```
{*X,
 {state = body.methods.inc(body.state),
  methods = body.methods}}
```

那么将肯定产生一个错误。例如，类型：

```
{∃X, {state:X, methods:{get:X→Nat,inc:X→X}, junk:Bool}}
```

是 Counter 的一个子类型。但上面的这一个包并没有这个类型：它缺少 junk 字段。所以才会有对 Counter 的任意子类型 C，上面的 sendinc 函数体“确实”有类型 C 这样的结果，除非类型规则自己能明白过来。事实上，很明显（例如，通过为 $F_{<:}$ 请求一个标志模型——参见 Robinson 和 Tennent，1998）在纯 $F_{<:}$ 中，形为 $\forall$ C <: T. C→C 的类型只包含有恒等函数。

现在已经提出了几种方法来弥补 $F_{<:}$ 这个缺点。一种可能性是把 $F_{<:}$ 系统改成 $F^{\omega}_{<:}$ 系统，并且用高阶囿量词将更多改进过的类型传给类似 sendinc 这样的函数。另一种可能性是保留类型 $\forall$ C <: Counter. C→C，但要在语言中增加一些特征，利用这些特征来生成这一类型的有趣

成员。一个最终的可能是在语言中添加一些引用。但是,在第 27 章已经是这样做了;本章的目标是试试看在一个纯函数框架中可以得到什么,且如何去得到。

后面的研究综合了其中的两种方法:$F^{\omega}_{<:}$ 用来说明前几节中提到的对象类型上的量词问题,以及多态记录更新原语(将在 32.7 节中定义),用来说明处理实例变量(参见 32.8 节)中出现的相关问题。

32.4　接口类型

使用类型操作子,可将 Counter 表达为两部分的组合体:

```
Counter = Object CounterM;
```

其中:

```
CounterM = λR. {get: R→Nat, inc:R→R};
```

是分类 $*\Rightarrow*$ 的类型操作子,表示 Counter 对象具体的方法接口,且:

```
Object = λM::*⇒*. {∃X, {state:X, methods:M X}};
```

是分类$(*\Rightarrow*)\Rightarrow*$的类型操作子,它具有所有对象类型的公用结构。通过再形式化,我们得到的是,将允许子类型化的可变部分(如方法接口)与不允许子类型化以避免其妨碍再打包的对象(如存在打包,状态与方法序对等)不变框架分开来。

我们需要类型操作子之上的囿量词来达到这种分割,因为它允许我们将方法接口从一个对象类型中抽出,尽管这些接口通过在 X 抽象方法接口本身,涉及到存在量词的边界状态类型 X。接口也因此而生成一个"参数化的参数"。这里参数可迭代的特性可在 Object 分类中和应用 Object CounterM 被简化的步骤中反映出来:首先,CounterM 被代换到 Object 的主体中,产生:

```
{∃X, {state:X, methods:(λR. {get: R→Nat, inc:R→R}) X}}
```

然后 X 被替换到 CounterM 的主体中,生成:

```
{∃X, {state:X, methods:{get:X→Nat,inc:X→X}}}.
```

如果我们按照相同的方法分割 ResetCounter:

```
ResetCounterM = λR. {get: R→Nat, inc:R→R, reset:R→R};
ResetCounter = Object ResetCounterM;
```

那么不仅像以前一样,有:

```
ResetCounter <: Counter
```

而且根据上面讨论的在类型操作子之间进行子类型化的规则有:

```
ResetCounterM <: CounterM
```

即,我们从对象类型分离到通用模板文件,加上一个特殊的接口,就给出了一个十分有意义的接口子类型化的概念,这些接口是从全部对象类型间的子类型化关系中分离出来的。

接口子类型化非常接近于(概念上和技术上)Bruce 等(1997)提出的匹配思想,Abadi 和 Cardelli(1995;1996)也深入研究过这一问题。

32.5 向对象发送消息

现在,我们可以通过抽象 CounterM 的子接口,而不是 Counter 的子类型来补充 32.3 节中 sendinc 未完成的定义:

```
sendinc =
  λM<:CounterM. λc:Object M.
    let {X, b} = c in
      {*X,
       {state = b.methods.inc(b.state),
        methods = b.methods}}
      as Object M;
```

```
▸ sendinc : ∀M<:CounterM. Object M → Object M
```

直觉上,sendinc 类型可以被读成"给我一个对象接口来改进计数器的接口,然后给我一个具有该接口的对象,我将返还给你一个有相同接口的另一个对象"。

32.5.1 练习[★]:为什么这个 sendinc 是良类型的而前一个不是?

为调用计数器的方法和重置计数器对象,我们利用适当的接口型构实例化多态方法调用函数,CounterM 或 ResetCounterM(假设 sendget 和 sendreset 已被类似地定义):

```
sendget [CounterM] (sendinc [CounterM] c);
```

```
▸ 6 : Nat
```

```
sendget [ResetCounterM]
          (sendreset [ResetCounterM]
             (sendinc [ResetCounterM] rc));
```

```
▸ 0 : Nat
```

32.5.2 练习[推荐,★★]:定义 sendget 和 sendreset。

32.6 简单的类

现在让我们来讨论类,从不带 self 的简单类开始(参见第 18 章)。

在 18.6 节,我们定义了一个简单的类(为命令式对象编码,当对象是方法的记录)作为一个从状态到对象的函数——一种生成多个包含有相同方法的对象方式,其中每种方法包含一个新分配的实例变量集合。在本章中,一个对象不再仅是方法的记录:它同时也包含了一个表示类型和一个状态。另一方面,由于这是一个纯函数模型,每一个方法都将状态作为一个参数(而且,如果需要的话,返回一个具有被更新过的状态对象),所以我们不需要在对象创建的时候传递一个状态给类。事实上,这里的一个类(我们仍然假设所有的对象都使用相同的表示类型)被简单地看做一个方法的记录:

```
counterClass =
   {get = λr:CounterR. r.x,
    inc = λr:CounterR. {x=succ(r.x)}}
  as {get: CounterR→Nat, inc:CounterR→CounterR};
```

```
▸ counterClass : {get:CounterR→Nat, inc:CounterR→CounterR}
```

或者，用 CounterM 操作子来更简洁地写出注释：

```
counterClass =
   {get = λr:CounterR. r.x,
    inc = λr:CounterR. {x=succ(r.x)}}
   as CounterM CounterR;

▸ counterClass : CounterM CounterR
```

我们将实例化这样的类，为状态提供一个初始值，并且将该方法的状态（即类）一起打包到一个对象中：

```
c = {*CounterR,
     {state = {x=0},
      methods = counterClass}}
    as Counter;

▸ c : Counter
```

定义一个子类只要建立一个新的方法记录，从以前已经定义过的类中复制出一些字段。

```
resetCounterClass =
   let super = counterClass in
   {get = super.get,
    inc = super.inc,
    reset = λr:CounterR. {x=0}}
   as ResetCounterM CounterR;

▸ resetCounterClass : ResetCounterM CounterR
```

为了概括这些简单的类，来处理第 18 章中提到的同样例子，还需要两样东西：在子类中添加新的实例变量的能力和对 self 的处理。下面的两节将解决第一个问题，32.9 节将讨论对 self 的处理，以结束本章。

32.7　多态更新

为了在类中添加实例变量，我们需要增加一个新的机制——在记录字段适当多态更新的原语和对记录类型的相应改进。在类之间允许实例变量变化意味着它们的子类实例变量可以实现超类的多态性。正是这一点，有了这些特征让我们来看看这是怎么发生的。

假设我们要定义一个 resetCounterClass 的子类，添加一个 backup 方法来保存计数器中的当前值，改变 reset 的行为使其恢复为这个被保存的值而不是初始的那个常值。为获得保存的值，我们需要将状态类型从{x:Nat}扩展到{x:Nat,old:Nat}。但表达方式的不同立即产生一个技术上的难题。定义 backupCounterClass 时，在 resetCounterClass 中重用 inc 方法的能力取决于该方法在两个类中是否有相同的行为。然而，如果实例变量的集合不同，那么它不可能有完全相同的行为：ResetCounter 中的 inc 需要一个类型为{x:Nat}的状态并返回一个相同类型的新状态，而 BackupCounter 的 inc 需要并产生类型为{x:Nat,old:Nat}的状态。

为了解决这一难题，注意到 inc 方法并不真的需要知道状态的类型是{x:Nat}还是{x:Nat,old:Nat}，只需要知道状态中包含一个实例变量 x。换句话说，我们可以通过提供类型 $\forall S <: \{x:Nat\}.S \to S$ 来将两种方法统一起来。

现在，这里出现了与第 32.3 节中含整个对象时带来的，由状态引起相同问题：在当前语言

中，类型 $\forall S <: \{x:Nat\}.S \to S$ 只能被恒等函数使用。为了解决这一难题，我们还需要一些机制，能提出一个更为精确的囿量词；这里最为直接的机制就是为记录字段的多态更新添加一个原语[①]。如果 r 是一个具有类型 T 的字段 x 的记录，且 t 是类型为 T 的一个项，那么写 $r \leftarrow x = t$ 来表示“除非它的 x 字段有值 t，这个记录看起来像 r ”。注意，这是更新操作的一个纯函数形式——它并不改变 r，但生成一个不同 x 字段的复制品。

通过使用这个记录更新原语，一个能够抓住 inc 方法体本意行为的函数可被粗略地展示如下：

```
f = λX<:{a:Nat}. λr:X. r←a = succ(r.a);
```

然而，还是必须谨慎一些。一种简单的更新操作子的类型化规则为：

$$\frac{\Gamma \vdash r : R \qquad \Gamma \vdash R <: \{l_j : T_j\} \qquad \Gamma \vdash t : T_j}{\Gamma \vdash r \leftarrow l_j = t : R}$$

但这个规则不合理。例如，假设有：

```
s = {x={a=5,b=6},y=true};
```

因为 $s:\{x:\{a:Nat,b:Nat\},y:Bool\}$，且 $\{x:\{a:Nat,b:Nat\},y:Bool\} <: \{x:\{a:Nat\}\}$，上面的规则得出：

```
s←x={a=8} : {x:{a:Nat,b:Nat},y:Bool},
```

这是错误的，因为 $s \leftarrow x = \{a = 8\}$ 归约为 $\{x = \{a = 8\}, y = true\}$。

这个问题的产生，是由于对 x 字段上使用深度子类型化而导出 $\{x:\{a:Nat,b:Nat\},y:Bool\} <: \{x:\{a:Nat\}\}$。深度子类型化将不允许在能被更新的字段中使用。为做到这一点，只要为这样的字段注释上一个特殊的标记，写为 # 。

在图 32.1 中给出了这些“可被更新的记录”的和更新操作规则本身。我们改进了记录类型的语法来使每个字段都可以被注释上一个变化的标签，用来指示是否允许深度子类型化。# 表示禁止在这些字段上进行子类型化，同时空的字符串则表示允许(在这里选择空串意思是未被标记的记录将和它们以前一样处理)。深度子类型化规则 S-RcdDepth 经过修改允许只对没有标记的字段中进行子类型化。最后，我们添加一条子类型化规则 S-RcdVariance，来允许字段中的标记从 # 变为空字符串——换句话说，来“忘记”给定的字段是可更新的。为更新原语建立的规则要求被替换的字段标记上 # 。E-Update 规则实现了更新操作。

上面的函数 f 现在可被写为：

```
f = λX<:{#a:Nat}. λr:X. r←a = succ(r.a);
▸ f : ∀X<:{#a:Nat}. X → X
```

且如这样使用：

```
r = {#a=0, b=true};
f [{#a:Nat,b:Bool}] r;
▸ {#a=1, b=true} : {#a:Nat, b:Bool}
```

---

① 和以前一样，有几种获得相同效果的方法——通过提出不同的原语(在 32.10 节中列出了一些)，或通过使用多态性，如在 Pierce 和 Turner(1994)中描述的那样——更加繁杂但理论上更具基本的选择方式。这里选择的原因是它简单，而且非常适合下面的例子。

更新操作的可靠性依赖于下面被改进后的子类型关系的结果:

32.7.1 事实: 如果$\vdash$ R <: {#l:T_1},那么 R = {⋯ #l:R_1⋯}其中$\vdash$ R_1 <:T_1 和$\vdash$ T_1 <:R_1。

32.7.2 练习[推荐,★★★]: 这个演算有最小的类型化性质吗?如果有就证明。如果没有,想办法补充一下。

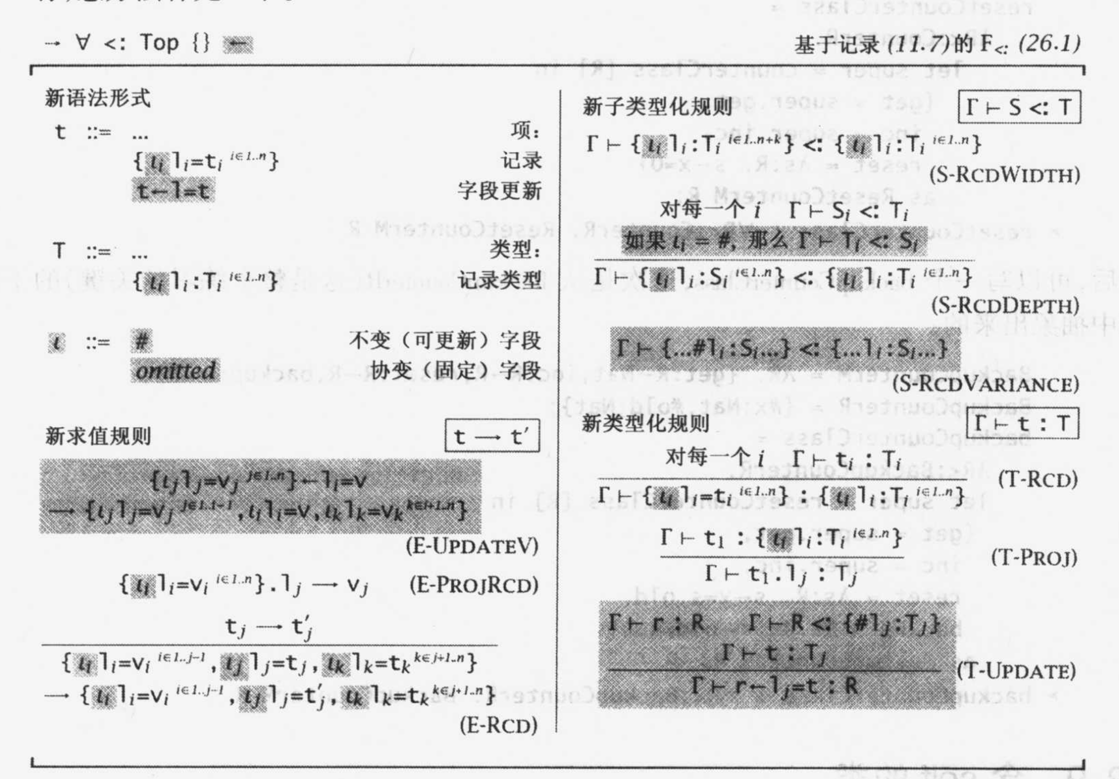


图 32.1 多态更新

32.8 添加实例变量

用前一节中的特征,写一个内部状态类型为多态的 counterClass:

```
CounterR = {#x:Nat};

counterClass =
   λR<:CounterR.
     {get = λs:R. s.x,
      inc = λs:R. s←x=succ(s.x)}
     as CounterM R;
```

```
▸ counterClass : ∀R<:CounterR. CounterM R
```

为了在新的 counterClass 中生成对象,简单地将 CounterR 写为如下表示类型:

```
c = {*CounterR,
      {state = {#x=0},
       methods = counterClass [CounterR]}}
    as Object CounterM;
```

```
▸ c : Counter
```

注意新类中创建的对象有相同的类型 Counter = Object CounterM,就像前面的对象一样:对实例变量处理的改变也完全是类内部的事。上面的方法调用函数也仍然可以用在新类中实例化的对象中。

我们可以将 resetCounterClass 写成同样的风格:

```
resetCounterClass =
   λR<:CounterR.
     let super = counterClass [R] in
       {get = super.get,
        inc = super.inc,
        reset = λs:R. s←x=0}
       as ResetCounterM R;
▸ resetCounterClass : ∀R<:CounterR. ResetCounterM R
```

最后,可以写一个 backupCounterClass,这次是从 BackupCounterR(这是整个练习的关键)的子类型中抽象出来的:

```
BackupCounterM = λR. {get:R→Nat,inc:R→R,reset:R→R,backup:R→R};
BackupCounterR = {#x:Nat,#old:Nat};
backupCounterClass =
   λR<:BackupCounterR.
   let super = resetCounterClass [R] in
    {get = super.get,
     inc = super.inc,
     reset = λs:R. s←x=s.old,
     backup = λs:R. s←old=s.x}
    as BackupCounterM R;
▸ backupCounterClass : ∀R<:BackupCounterR. BackupCounterM R
```

32.9 含 self 的类

在 18.9 节中,我们谈到了命令式类,它允许类中的方法互递归,并且还谈到了如何扩展这种类。这种扩展在纯函数框架中仍然是合理的。

首先讨论适合相同表示类型 R 的 self 方法集 counterClass 进行的抽象:

```
counterClass =
   λR<:CounterR.
   λself: Unit→CounterM R.
   λ_:Unit.
      {get = λs:R. s.x,
       inc = λs:R. s←x=succ(s.x)}
      as CounterM R;
```

和 18.9 节一样,类的 Unit 参数用于推迟生成一个对象方法 fix 操作中的求值。self 的类型包括一个匹配的 Unit 抽象。

为了在这个类中建立一个对象,我们取 counterClass 函数的不动点并应用到 unit 上:

```
c = {*CounterR,
      {state = {#x=0},
       methods = fix (counterClass [CounterR]) unit}}
     as Object CounterM;

▸ c : Counter
```

接着定义一个提供 set 操作的子类,并且具有下面的接口:

```
SetCounterM = λR. {get: R→Nat, set:R→Nat→R, inc:R→R};
```

SetCounterClass 的实现定义了一个 set 方法,并在它的 inc 方法的实现中用到 self 中 set 和 get 方法:

```
setCounterClass =
   λR<:CounterR.
   λself: Unit→SetCounterM R.
   λ_:Unit.
      let super = counterClass [R] self unit in
      {get = super.get,
       set = λs:R. λn:Nat. s←x=n,
       inc = λs:R. (self unit).set s (succ((self unit).get s))}
   as SetCounterM R;
```

最后,将本章中的所有机制结合在一起,可以创建一个仪器计数器的子类,该子类中的 set 操作可以统计它被调用的次数:

```
InstrCounterM =
  λR. {get: R→Nat, set:R→Nat→R, inc:R→R, accesses:R→Nat};

InstrCounterR = {#x:Nat,#count:Nat};

instrCounterClass =
   λR<:InstrCounterR.
   λself: Unit→InstrCounterM R.
   λ_:Unit.
      let super = setCounterClass [R] self unit in
      {get = super.get,
     set = λs:R. λn:Nat.
             let r = super.set s n in
             r←count=succ(r.count),
     inc = super.inc,
     accesses = λs:R. s.count}
   as InstrCounterM R;
```

注意,因为 inc 的实现是根据 self 中的 set 方法,对 inc 的调用次数也算在访问计数中。

这里为实现包装,创建一个仪器计数器对象,并给它发送消息:

```
ic = {*InstrCounterR,
       {state = {#x=0,#count=0},
        methods = fix (instrCounterClass [InstrCounterR]) unit}}
      as Object InstrCounterM;

▸ ic : Object InstrCounterM

sendaccesses [InstrCounterM] (sendinc [InstrCounterM] ic);

▸ 1 : Nat
```

32.9.1 练习[推荐,★★★]:定义一个添加 backup 和 reset 方法的 instrCounterClass 的子类。

32.10 注释

在类型的 lambda 演算中,对象的第一个"纯函数"解释是基于递归定义的记录;最初被 Cardelli(1984)提出,而 Kamin 和 Reddy(Reddy,1988;Kamin 和 Reddy,1994),Cook 和 Palsberg(1989)以及 Mitchell(1990a)研究了许多它的变化。在它的无类型形式中,这种模型应用于无类型的面向对象语言的指称语义是非常有效的。在它的类型化形式中,可以被用于对单独面向对象的例子进行编码,但会造成类型化面向对象语言中统一解释上的难题。在这个方面,其最成功的是 Cook 和他的同伴(Cook,Hill 和 Canning,1990;Canning,Cook,Hill 和 Olthoff,1989a;Canning,Cook,Hill,Olthoff 和 Mitchell,1989b)。

Pierce 和 Turner(1994)提出一种编码,只依赖于具有存在类型的类型系统,而与递归类型无关。这引导 Hofmann 和 Pierce(1995b)提出第一个在函数演算中,对象统一类型驱动的解释。在同一个会议上,Bruce 发表了一篇关于函数面向对象语言语义方面的论文。这个语义最初被作为一个能直接映射到 $F^{\omega}_{<:}$ 的指称模型,而最近又被重新形式化为一个依赖于存在和递归类型的对象编码方式。同时,由于在 lambda 演算中解决编码对象的困难,Abadi 和 Cardelli 提出了一种原始对象的演算(1996)。然而后来,Abadi,Cardelli 和 Viswanathan(1996)根据囿存在量词和递归类型,发现对象演算的一种可靠的编码。Bruce 等(1999)与 Abadi 和 Cardelli(1996)调查了这些发展情况。

本章中的对象编码已经被扩展到包含多重继承(拥有多个超类的类)——根据 Compagnoni 和 Pierce(1996)。关键的技术思想是用交叉类型(参见 15.7 节)对的 $F^{\omega}_{<:}$ 的扩展。

现在已经有多种方案用于解决在 32.3 节中观察的纯二阶囿量词的缺点。除了在本章中见到的两个——高阶囿量词和多态记录更新原语以外,还有其他几种多态记录更新的样式(Cardelli 和 Mitchell,1991;Cardelli,1992;Fisher 和 Mitchell,1996;Poll,1996),例如递归类型的结构化展开(Abadi 和 Cardelli,1996),正子类型化(Hofmann 和 Pierce,1995a),对存在类型的多态重包装(Pierce,1996)和类型析构子(Hofmann 和 Pierce,1998)等。

Wand(1987,1988,1989b),Rémy(1990,1989,1992),Vouillon(2000,2001)和其他一些人发展了一种基于行变量多态性的不同解决方向,并且组成了 OCaml 的面向对象的特点基础(Rémy 和 Vouillon,1998)。

"在开始处开始,"国王十分严肃地说,"继续进行直到终点:然后停止。"

—— Lewis Carroll

附录 A　部分习题解答

3.2.4　解答：$|S_{i+1}| = |S_i|^3 + |S_i| \times 3 + 3$，且 $|S_0| = 0$。所以 $|S_3| = 59439$。

3.2.5　解答：直接归纳证明来解决这一问题。当 $i = 0$ 时，不需证明。接下来，当 $j \geq 0$ 时假定 $i = j + 1$ 和 $S_j \subseteq S_i$，需要证明出 $S_i \subseteq S_{i+1}$，也就是对于任何一个项 $t \in S_i$，需要证明 $t \in S_{i+1}$。对于假设有 $t \in S_i$。根据由三个集合组成的联合 S_i 的定义，我们知道 t 必有以下三种情况之一：

1. $t \in \{true, false, 0\}$。在这种情况中，根据 S_{i+1} 的定义，显然得到 $t \in S_{i+1}$。
2. t = succ t_1，pred t_1 或 iszero t_1，其中 $t_1 \in S_j$。根据归纳假设有 $S_j \subseteq S_i$，可得到 $t_1 \in S_i$，所以根据 S_{i+1} 的定义得到 $t \in S_{i+1}$。
3. t = if t_1 then t_2 else t_3，当 $t_1, t_2, t_3 \in S_j$。同样根据归纳假设有 $S_j \subseteq S_i$，根据 S_{i+1} 的定义得到 $t \in S_{i+1}$。

3.3.4　解答：（仅给出对深度归纳的讨论；其他情况都是类似的）。我们知道对于每一项 s，如果对所有深度小于 s 的 r 有 $P(r)$，于是有 $P(s)$；现在我们必须证明对所有的 s 有 $P(s)$。定义一个新的自然数上的谓词 Q 如下所示：

$$Q(n) \quad = \quad \forall s \text{ with } depth(s) = n.\ P(s)$$

现在利用自然数归纳(2.4.2)来证明对所有的 n 有 $Q(n)$。

3.5.5　解答：假设 P 是一个在求值语句上推导的谓词。
如果，对于每一个推导 $\mathcal{D}$，
对于所有直接子推导 $\mathcal{C}$ 给出 $P(\mathcal{C})$
我们能证明 $P(\mathcal{D})$，
于是对所有 $\mathcal{D}$ 有 $P(\mathcal{D})$。

3.5.10　解答：

$$\frac{t \longrightarrow t'}{t \longrightarrow^* t'}$$

$$t \longrightarrow^* t$$

$$\frac{t \longrightarrow^* t' \qquad t' \longrightarrow^* t''}{t \longrightarrow^* t''}$$

3.5.13　解答：(1)3.5.4 和 3.5.11 失败。3.5.7，3.5.8 和 3.5.12 仍为有效；(2)现在仅 3.5.4 失败，其余都有效。在第(2)部分有趣的是，尽管一步求值在这一规则之下变为非确定的，最终的多步求值结果仍是可确定的：条条大路通罗马。确实，这一论证的严格证明并不十分困难，尽管不像以前的那样微不足道。主要的观察在于一步求值的关系有所谓的菱形性质之称：

A.1 引理[菱形性质]:如果有 $r \rightarrow s$ 和 $r \rightarrow t$,且 $s \neq t$,于是有一些项 u 满足 $s \rightarrow u$ 和 $t \rightarrow u$。

证明:从求值规则出发,显然这种情况仅当 r 具有 if r_1 then r_2 else r_3 的形式时才会出现。根据对推导出 $r \rightarrow s$ 和 $r \rightarrow t$ 的推导序对的归纳,及一个关于两个推导中的最后规则的情况分析,有:

情况 i:

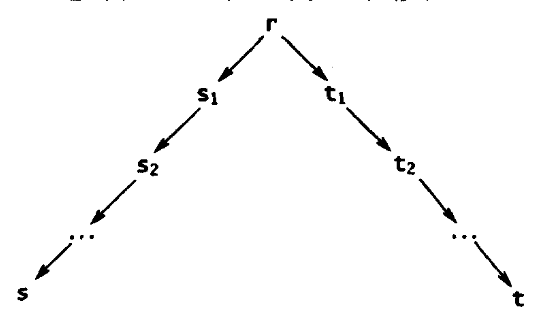
假设根据 E-IfTrue 有 $r \rightarrow s$ 且根据 E-Funny2 有 $r \rightarrow t$。于是,从这些规则的形式出发,我们知道 $s = r_2$ 且 $t =$ if true then r_2' else r_3,其中 $r_2 \rightarrow r_2'$。但接下来选择 $u = r_2'$ 给了我们所需要的,因为我们知道 $s \rightarrow r_2'$ 且通过 E-IfTrue 可得到 $t \rightarrow r_2'$。

情况 ii:

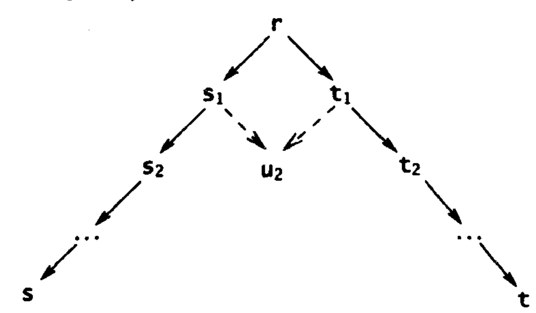
假设在推导 $r \rightarrow s$ 和 $r \rightarrow t$ 中的最终规则是 E-If。通过 E-If 的形式,我们知道 s 一定具备 if r_1' then r_2 else r_3 的形式,t 必有形式 if r_1'' then r_2 else r_3,其中 $r_1 \rightarrow r_1'$ 且 $r_1 \rightarrow r_1''$。但这时根据归纳假设,有一些满足 $r_1' \rightarrow r_1'''$ 和 $r_1'' \rightarrow r_1'''$ 的 r_1''' 项,可以通过讨论 $u =$ if r_1''' then r_2 else r_3 和通过 E-If 观察 $s \rightarrow u$ 和 $t \rightarrow u$ 来完成这种情况的讨论。

这种讨论对于其他情况是类似的。

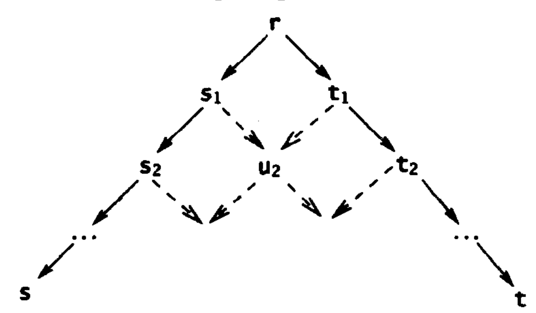
这种结论的惟一性证明可直接得出一个"尾随图"。假设 $r \rightarrow^* s$ 且 $r \rightarrow^* t$。



接下来可以使用引理 A.1 将 s_1 和 t_1"拉到一起":



再使用它来将 s_2 和 u_2 拉到一起,然后是 u_2 和 t_2:



等等，一直进行下去直到将 s 和 t 拉到一起，这时一步菱形形成了一个完整的大菱形。

它说明了如果 r 可被求值为 v，也可被求值为 w，则 v 与 w 必定相同（它们都是范式，只要从相同的形式开始求值，就只能得到相同的结果）。

3.5.14　解答： 对 t 的结构进行归纳。

情况：t 是一个值。

由于每一个值都有范式，这一情况不可能出现。

情况： $t = \text{succ } t_1$

根据求值规则，我们发现只有规则 E-Succ 可能被用于推导 $t \to t'$ 和 $t \to t''$（其他所有的规则的左端最外端的构造子不是 succ）。所以必定存在两个子推导，其结论为 $t_1 \to t_1'$ 和 $t_1 \to t_1''$。通过归纳假设（由于 t_1 是 t 的子项），我们得到 $t_1' = t_1''$。但接下来有 $\text{succ } t_1' = \text{succ } t_1''$。

情况： $t = \text{pred } t_1$

这里有三个求值规则（E-Pred，E-PredZero 和 E-PredSucc），这些可能已经被用来将 t 归约为 t′和 t″。注意，这些规则不重叠：如果 t 与一个规则的左端匹配，那么它肯定不会与其他规则左端匹配（例如，如果 t 与 E-Pred 匹配，那么 t_1 肯定不是一个值，尤其不是 0 和 succ v）。这就告诉我们同余规则一定已经用于推导 $t \to t'$ 和 $t \to t''$。如果这个规则是 E-Pred，那么像以前的情况那样使用归纳假设。如果这是 E-PredZero 或 E-PredSucc 规则，那么结果直接可得出。

情况：其他情况

类似。

3.5.16　解答： 让我们使用元变量 t 来包括新的由 wrong 扩展项的新集合（包括所有以 wrong 为子短语的项），g 来包括最初不包括 wrong 的"good"项初始集。写含 wrong 转换的新求值关系为 $t \xrightarrow{w} t'$，并写出求值的初始形式 $g \xrightarrow{o} g'$。现在，这两种方式可形式化描述为：任何（初始的）在最初的语义下求值受阻的项将会在新的语义中的求值结果都为 wrong，反之亦然。正式地：

A.2　命题： 对所有初始的项 g，（关系写出 $g \xrightarrow{o}{}^{*} g'$，其中 g′受阻式）当且仅当（$g \xrightarrow{w}{}^{*}$ wrong）。

要证明这个命题，分多步进行。首先，注意到我们已加入的新转换在定理（3.5.14）中不是无效的。

A.3　引理： 被扩展的求值关系是可确定的。

这意味着在就初始语义中无论何时 g 能一步到 g′，它都能在被扩展的语义中到 g′，此外 g′是 g 在新语义中惟一能到达的项。

接下来，我们显示一个已经在初始语义中受阻的项总能在被扩展的语义中求值为 wrong。

A.4　引理： 如果 g 受阻，那么 $g \xrightarrow{w}{}^{*}$ wrong。

证明： 对 g 的结构进行归纳。

情况: g = true, false, 或 0

不可能发生(我们假定 g 受阻)。

情况: g = if g_1 then g_2 else g_3

由于 g 受阻, g_1 必定是范式(否则 E-If 适用)。显然, g_1 不可能为 true 或 false(否则 E-IFtrue 或 E-IfFalse 中的一个会适用且 g 不会受阻)。考虑到剩余的情况:

子情况: g_1 = if g_{11} then g_{12} else g_{13}

因为 g_1 是范式而显然不是值,它会受阻由归纳假设可求得 $g_1 \xrightarrow{w}^* $ wrong。由此,我们能够构造推导 $g \xrightarrow{w}^*$ if wrong then g_2 else g_3。如需要的那样,增加一个规则 E-If-Wrong 的最终实例生成 $g \xrightarrow{w}^*$ wrong。

子情况: g_1 = succ g_{11}

如果 g_{11} 是一个数值,那么 g_1 是一个 badbool,而且规则 E-If-Wrong 立即生成 $g \xrightarrow{w}$ wrong。否则,根据定义 g_1 受阻,由归归纳假设可得 $g_1 \xrightarrow{w}^*$ wrong。从这一推导,我们创建了一个推导 $g \xrightarrow{w}^*$ if wrong then g_2 else g_3。增加一个规则 E-If-Wrong 的最终实例生成 $g \xrightarrow{w}^*$ wrong。

其他子情况:

类似。

情况: g = succ g_1

由于 g 受阻,我们(由值定义)得知 g_1 必须是范式而不是一个数值。有两种可能:要么 g_1 是 true 或 false,要么 g_1 自身不是值,那么受阻。在第一种情况,规则 E-Succ-Wrong 直接生成 $g \xrightarrow{w}^*$ wrong;在第二种情况,归纳假设得出 $g_1 \xrightarrow{w}^*$ wrong,按以前那样继续下去。

其他情况:

类似。

引理(A.3)和引理(A.4)一起给命题(A.2)的"仅当"(⇒)这部分证明。对另一半,需要证明在被扩充的语义中"将要出错"的项在初始的语义中受阻。

A.5 引理:如果在被扩展的语义中 $g \xrightarrow{w} t$ 和 t 将 wrong 作为一个子项包含在内,那么 g 在初始的语义中受阻。

证明:在(已讨论的)求值推导中直接归纳。

与引理(A.3)结合,可证命题(A.2)的"当且"(⇐)的一半,证明结束。

3.5.17 解答:分别在命题(A.7)和命题(A.9)中论证了"当且仅当"两个方向,对于每一种情况,我们由技术引理开始建立了一些有用的多步(或单步)的求值性质。

A.6 引理:如果 $t_1 \rightarrow^* t_1'$ 那么 if t_1 then t_2 else $t_3 \rightarrow^*$ if t_1' then t_2 else t_3(对于其他项构造子这也是类似的)。

证明:简单归纳。

A.7 命题:如果 $t \Downarrow v$ 那么 $t \rightarrow^* v$

证明:根据归纳于 t ⇓ v 的推导,对最后规则中进行情况分析。

情况 B-VALUE: t = v

立即可证。

情况 B-IFTRUE: t = `if` t_1 `then` t_2 `else` t_3
t_1 ⇓ `true`
t_2 ⇓ v

根据归纳假设,$t_1 \rightarrow^* $ true。根据引理(A.6):

`if` t_1 `then` t_2 `else` t_3 $\rightarrow^*$ `if true then` t_2 `else` t_3.

根据 E-IfTrue,if true then t_2 else $t_3 \rightarrow t_2$。根据归纳假设,$t_2 \rightarrow^*$ v。结果可从$\rightarrow^*$的转换得出。

其他情况:

类似。

A.8 引理:如果:

`if` t_1 `then` t_2 `else` t_3 $\rightarrow^*$ v,

于是要么 $t_1 \rightarrow^*$ true 且 $t_2 \rightarrow^*$ v,要么 $t_1 \rightarrow^*$ false 且 $t_3 \rightarrow^*$ v。此外,t_1 和 t_2 或 t_3 的求值序列肯定会比给出的求值序列短(对于其他项构造子也是类似的)。

证明:根据归纳于已给定的求值序列长度。由于一个条件不是值,故至少存在一步求值。通过在这一步的最后规则中的情况分析(注意必须有一个 If 规则)可知:

情况 E-IF: `if` t_1 `then` t_2 `else` t_3 $\rightarrow$ `if` t_1' `then` t_2 `else` t_3 $\rightarrow^*$ v
$t_1 \rightarrow t_1'$

根据假设归纳,要么 $t_1' \rightarrow^*$ true 和 $t_2 \rightarrow^*$ v,要么 $t_1' \rightarrow^*$ false 和 $t_3 \rightarrow^*$ v。将最初的步骤 $t_1 \rightarrow t_1'$ 加入到 $t_1' \rightarrow^*$ true 或或 $t_1' \rightarrow^*$ false 的推导中产生想要的结果。很容易检查出最后的求值序列比原来的短。

情况 E-IFTRUE: `if true then` t_2 `else` t_2 $\rightarrow$ t_2 $\rightarrow^*$ v

立即可证。

情况 E-IFFALSE: `if false then` t_2 `else` t_2 $\rightarrow$ t_3 $\rightarrow^*$ v

立即可证。

A.9 命题:如果 $t \rightarrow^*$ v 那么 t ⇓ v 成立。

证明:根据归纳于在给定的 $t \rightarrow^*$ v 推导中小步求值的步骤数。

如果在 0 步 $t \rightarrow^*$ v,那么 t = v 且由 B-Value 得出结果。否则,通过 t 形式的情况分析可知:

情况: t = `if` t_1 `then` t_2 `else` t_3

根据引理(A.8),或者(1)$t_1 \rightarrow^*$ true 和 $t_2 \rightarrow^*$ v 或者(2)$t_1 \rightarrow^*$ false 和 $t_3 \rightarrow^*$ v。两种情况的讨论是相似的,所以假定有情况(1)。引理(A.8)也告诉我们 $t_1 \rightarrow^*$ true 和 $t_2 \rightarrow^*$ v 的求值序列比 t 给出的短,所以应用归纳假设,得出 t_1 ⇓ true 和 t_2 ⇓ v。以此,我们能使用规则 B-IfTrue 来推导 t ⇓ v。

对于其他的项构造子都是相似的。

4.2.1 解答: 每次 eval 调用它自身,它增加一个 try 句柄到调用堆栈。由于每步求值都有一个递归调用,堆栈最终会溢出。其实,在一个 try 中将递归调用包装到 eval 中意味着没有尾调用,尽管它看起来相像。比较好(但不易读)的 eval 版本是:

```
let rec eval t =
  let t'opt = try Some(eval1 t) with NoRuleApplies → None in
  match t'opt with
    Some(t') → eval t'
  | None → t
```

5.2.1 解答:

```
or = λb. λc. b tru c;
not = λb. b fls tru;
```

5.2.2 解答:

```
scc2 = λn. λs. λz. n s (s z);
```

5.2.3 解答:

```
times2 = λm. λn. λs. λz. m (n s) z;
```

或者,更简洁地:

```
times3 = λm. λn. λs. m (n s);
```

5.2.4 解答: 再次,有多种方式来做:

```
power1 = λm. λn. m (times n) c1;
power2 = λm. λn. m n;
```

5.2.5 解答:

```
subtract1 = λm. λn. n prd m;
```

5.2.6 解答: 求值 prd c_n 需要 $O(n)$ 步,由于 prd 使用 n 来构造 n 个数序对序列,那么选择这一序列最后一对的第一个分量。

5.2.7 解答: 这是一个简单的解答:

```
equal = λm. λn.
          and (iszro (m prd n))
              (iszro (n prd m));
```

5.2.8 解答: 这是我的想法:

```
nil = λc. λn. n;
cons = λh. λt. λc. λn. c h (t c n);
head = λl. l (λh.λt.h) fls;
tail = λl.
         fst (l (λx. λp. pair (snd p) (cons x (snd p)))
                (pair nil nil));
isnil = λl. l (λh.λt.fls) tru;
```

还有一个不同的方法:

```
nil = pair tru tru;
cons = λh. λt. pair fls (pair h t);
head = λz. fst (snd z);
tail = λz. snd (snd z);
isnil = fst;
```

5.2.9　解答:我们使用 if 而不是 test 来阻止条件句的两个分支总是被求值,因为它会使 factorial 发散。当使用 test 时,为了防止这种发散,需要通过将它们包装在虚拟 lambda 抽象中来保护两个分支。由于抽象是数值,值调用的求值策略看起来不受它们的支配,而是逐字地将它们传送到 test 中,供 test 选择一个并把它返回。我们于是将整个 test 表达式应用到一个虚参数,例如 c_0 来强制选择分支的求值:

```
ff = λf. λn.
       test
         (iszro n) (λx. c1) (λx. (times n (f (prd n)))) c0;
factorial = fix ff;
equal c6 (factorial c3);
▸ (λx. λy. x)
```

5.2.10　解答:这里有一个递归函数来完成这一工作:

```
cn = λf. λm. if iszero m then c0 else scc (f (pred m));
churchnat = fix cn;
```

它进行的快速检查为:

```
equal (churchnat 4) c4;
▸ (λx. λy. x)
```

5.2.11　解答:

```
ff = λf. λl.
       test (isnil l)
         (λx. c0) (λx. (plus (head l) (f (tail l)))) c0;
sumlist = fix ff;

l = cons c2 (cons c3 (cons c4 nil));
equal (sumlist l) c9;
▸ (λx. λy. x)
```

一个列表求和函数当然也可以不使用 fix 来写:

```
sumlist' = λl. l plus c0;
equal (sumlist l) c9;
▸ (λx. λy. x)
```

5.3.3　解答:根据对 t 的长度归纳。假定项的期望值小于 t,必须证明 t 自身;如成功,能够总结出对所有 t 这种值都支持。这里有三种情况需要考虑:

情况:　$t = x$

直接:$|FV(t)| = |\{x\}| = 1 = size(t)$。

情况:　$t = \lambda x.t_1$

根据归纳假设，$|FV(t_1)| \leq size(t_1)$。现在计算如下：$|FV(t)| = |FV(t_1) \setminus \{x\}| \leq |FV(t_1)| \leq size(t_1) < size(t)$。

情况: $t = t_1\ t_2$

根据归纳假设：$|FV(t_1)| \leq size(t_1)$ 和 $|FV(t_2)| \leq size(t_2)$。现在计算如下 $|FV(t)| = |FV(t_1) \cup FV(t_2)| \leq |FV(t_1)| + |FV(t_2)| \leq size(t_1) + size(t_2) < size(t)$。

5.3.6 解答: 对于完全(非确定性的)β 归约，规则是：

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{t_1\ t_2 \longrightarrow t_1\ t_2'} \qquad \text{(E-App2)}$$

$$(\lambda x.t_{12})\ t_2 \longrightarrow [x \mapsto t_2]t_{12} \qquad \text{(E-AppAbs)}$$

(注意：值的语法范畴没有被使用)。

对于规范序的方法，一种书写规则是：

$$\frac{na_1 \longrightarrow na_1'}{na_1\ t_2 \longrightarrow na_1'\ t_2} \qquad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{nanf_1\ t_2 \longrightarrow nanf_1\ t_2'} \qquad \text{(E-App2)}$$

$$\frac{t_1 \longrightarrow t_1'}{\lambda x.t_1 \longrightarrow \lambda x.t_1'} \qquad \text{(E-Abs)}$$

$$(\lambda x.t_{12})\ t_2 \longrightarrow [x \mapsto t_2]t_{12} \qquad \text{(E-AppAbs)}$$

其中范式的语法范畴，非抽象范式和非抽象被定义如下：

```
nf   ::=                                范式:
         λx.nf
         nanf
nanf ::=                                非抽象范式:
         x
         nanf nf
na   ::=                                非抽象:
         x
         t1 t2
```

(比起其他定义，这个定义有些难理解。规范序归约可以这样来定义“除非 选择最左或最外的约式，它就类似于全 β 归约”)。

懒惰策略定义值为任意的抽象——与值调用相同。求值规则为：

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \text{(E-App1)}$$

$$(\lambda x.t_{12})\ t_2 \longrightarrow [x \mapsto t_2]t_{12} \qquad \text{(E-AppAbs)}$$

5.3.8　解答：

$$\frac{}{\lambda x.t \Downarrow \lambda x.t}$$

$$\frac{t_1 \Downarrow \lambda x.t_{12} \qquad t_2 \Downarrow v_2 \qquad [x \mapsto v_2]t_{12} \Downarrow v}{t_1\ t_2 \Downarrow v}$$

6.1.1　解答：

```
c0 = λ. λ. 0;
c2 = λ. λ. 1 (1 0);
plus = λ. λ. λ. λ. 3 1 (2 0 1);
fix = λ. (λ. 1 (λ. (1 1) 0)) (λ. 1 (λ. (1 1) 0));
foo = (λ. (λ. 0)) (λ. 0);
```

6.1.5　解答：两个函数可以被定义如下：

$\mathit{removenames}_\Gamma(x)$	=	Γ 中最右边的 x 的索引
$\mathit{removenames}_\Gamma(\lambda x.t_1)$	=	$\lambda.\,\mathit{removenames}_{,x}(t_1)$
$\mathit{removenames}_\Gamma(t_1\ t_2)$	=	$\mathit{removenames}_{\Gamma,x}(t_1)\ \mathit{removenames}_{\Gamma,x}(t_2)$
$\mathit{removenames}_\Gamma(k)$	=	Γ 中第 k 个名字
$\mathit{restorenames}_\Gamma(\lambda.t_1)$	=	$\lambda.\,\mathit{restorenames}_{\Gamma,x}(t_1)$
		其中 x 是不属于 dom(Γ)中的第一个名字
$\mathit{restorenames}_\Gamma(t_1\ t_2)$	=	$\mathit{restorenames}_{\Gamma,x}(t_1)\ \mathit{restorenames}_{\Gamma,x}(t_2)$

removenames 和 *restorenames* 要求的特性可直接通过对项的结构归纳来证明。

6.2.2　解答：

1. λ.λ.1(0 4)
2. λ. 0 3(λ0 1 4)

6.2.5　解答：

`[0 ↦ 1] (0 (λ.λ.2))`	=	`1 (λ. λ. 3)`
	即，	`a (λx. λy. a)`
`[0 ↦ 1 (λ.2)] (0 (λ.1))`	=	`(1 (λ.2)) (λ.(2 (λ.3)))`
	即，	`(a (λz.a)) (λx.(a (λz.a)))`
`[0 ↦ 1] (λ. 0 2)`	=	`λ. 0 2`
	即，	`λb. b a`
`[0 ↦ 1] (λ. 1 0)`	=	`λ. 2 0`
	即，	`λa'. a a'`

6.2.8　解答：如果 Γ 是一个命名上下文，将 Γ 中的索引 x 的索引写为 Γ(x)，从右边计算。现在，我们所希望得到的性质是：

$$\mathit{removenames}_\Gamma([x \mapsto s]t) = [\Gamma(x) \mapsto \mathit{removenames}_\Gamma(s)](\mathit{removenames}_\Gamma(t))$$

可通过使用定义(5.3.5)和定义(6.2.4)，一些简单计算和一些关于 *removenames* 简单引理以及其他项上的基本操作，来对 t 进行归纳。对于抽象情况，约定(5.3.4)起到了关键作用。

6.3.1　解答：一个索引为负的惟一方式，是标为 0 的变量是否出现在移位项中任何地方。但是这不可能发生，因为我们为变量 0 执行了一个代换(既然已用变量 0 进行了代换，变量 0 代换的项已经向上移位了，那么它显然不可能包含变量 0 的任何实例)。

6.3.2 解答:使用索引和层的描述等价性的证明能在 Lescanne 和 Rouyer-Degli(1995)中找到。de Bruijn 层被 de Bruijn(1972)和 Filinski(1999,5.2 节)讨论过。

8.3.5 解答:否:除去这一规则打破了进展性质。如果我们真的反对定义 0 的前驱,需要另一种方式处理它——例如,如果程序尝试它,则引发一个异常(参见第 14 章),或者通过改进 pred 类型来使它只能被合法用于正数,可能使用交叉类型(参见第 15.7 节)或依赖类型(参见 30.5 节)。

8.3.6 解答:这里有一个反例:项(if false then true else 0)是一个不良类型,但可求值为良类型项 0。

8.3.7 解答:大步语义的类型保持性质与我们给小步语义的性质类似:如果良类型项求值到一些最终值,那么这个值和初始项有相同的类型。证明与已经给出的相似。从另一方面,进展性质现在做出了更强的声明:每一个良类型项可以被求值到最终值——也就是说,对良类型项求值总会终止。对于算术表达式,这碰巧会发生,但对于更有趣的语言(包括普通递归的语言,参见 11.11 节)它常常不是真的。对于这些语言,我们没有进展性质:事实上,很难说清楚到达错误状态和失败终止之间的不同。这就是语言理论家们通常喜爱小步形式的一个原因。
一个不同的选择是给出明显的 wrong 转换的大步语义(如在练习 8.3.8 的形式中)。这种形式在 Abadi 和 Cardelli 的对象演算的操作语法中(Abadi 和 Cardelli,1996,p.87)使用过。

8.3.8 解答:在扩充的语义中根本没有受阻状态,每一个非值项或者以通常方法求值为其他项,或者直接为 wrong(当然这必须被证明),所以进展性质是平凡的。另一方面,主题归约定理现在告诉我们更多一些。既然 wrong 没有类型,那么一个良类型项只能求值到另一个良类型项和说明,尤其是,一个良类型项不可能一步到 wrong。实际上,老的进展定理的证明将成为新的保持证明的一部分。

9.2.1 解答:因为类型表达式集为空(类型语法中不存在基本情况)。

9.2.3 解答:一个这样的上下文是:

$$\Gamma = \texttt{f:Bool}\to\texttt{Bool}\to\texttt{Bool}, \texttt{x:Bool}, \texttt{y:Bool}.$$

通常,任一个形为:

$$\Gamma = \texttt{f:S}\to\texttt{T}\to\texttt{Bool}, \texttt{x:S}, \texttt{y:T}$$

的上下文,其中 S 和 T 是任意类型,都可达到这一要求。这种推理是对第 22 章中研究的类型重构算法的核心。

9.3.2 解答:假设项 x x 没有类型 T。于是,根据逆转引理,左端子项(x)必定有类型 $T_1 \to T_2$ 而且右端子项(同样是 x)必定有类型 T_1。使用逆转引理的变量情况,我们发现 $x : T_1 \to T_2$ 和 $x : T_1$ 必定都来自 Γ 中的假设。既然对 Γ 中的 x 只存在一个绑定,这意味着 $T_1 \to T_2 = T_1$。但这是不可能的,所有类型都有有限的长度,所以一个类型不可能作为它自身的子语出现,由此相互矛盾。
注意,如果类型允许无穷大,那么我们能说明等式 $T_1 \to T_2 = T_1$ 成立。在第 20 章详细地谈到这些。

9.3.3 解答: 假定 $\Gamma\vdash t:S$ 和 $\Gamma\vdash t:T$。我们对 $\Gamma\vdash t:T$ 推导进行归纳,得出 $S=T$。

情况 T-VAR: $t = x$
有 $x:T\in\Gamma$

根据逆转引理(9.3.1)的情况(1),$\Gamma\vdash t:S$ 的任何推导的最终规则都必须也是 T-Var 且 $S=T$。

情况 T-ABS: $t = \lambda y:T_2.t_1$
$T = T_2\to T_1$
$\Gamma, y:T_2 \vdash t_1 : T_1$

根据逆转引理的情况(2),$\Gamma\vdash t:S$ 的任何推导的最终规则都必须也是 T-Vabs,而且这个推导必有在 $S=T_2\to S_1$ 下结论为 $\Gamma,y:T_2\vdash t_1:S_1$ 的子推导。根据对结论为($\Gamma,y:T_2\vdash t_1:T_1$)的子推导进行归纳假设,我们得到 $S_1=T_1$,从中直接得出 $S=T$。

情况 T-APP, T-TRUE, T-FALSE, T-IF:

类似。

9.3.9 解答: 根据对 $\Gamma\vdash t:T$ 推导归纳。归纳中的每一步,假设所需的性质满足所有子推导(即,如果 $\Gamma\vdash s:S$ 和 $s\to s'$,那么 $\Gamma\vdash s':S$,只要 $\Gamma\vdash s:S$ 是由当前的子推导证明的)且通过对推导中用到的最后规则进行情况分析可得出:

情况 T-VAR: $t = x \quad x:T\in\Gamma$

不可能发生(没有一个将变量作为左端的求值规则)。

情况 T-ABS: $t = \lambda x:T_1.t_2$

不可能发生。

情况 T-APP: $t = t_1\ t_2 \quad \Gamma\vdash t_1 : T_{11}\to T_{12} \quad \Gamma\vdash t_2 : T_{11}$
$T = T_{12}$

查看图 9.1 左端有应用的求值规则,我们发现存在三个可以推导出 $t\to t'$ 的规则:E-App1,E-App2 和 E-AppAbs。对每种情况分别考虑。

子情况 E-APP1: $t_1 \to t_1' \quad t' = t_1'\ t_2$

由 T-App 情况的假设,得到一个结论为 $\Gamma\vdash t_1:T_{11}\to T_{12}$ 的初始的类型化推导的子推导。可以对这个子推导中应用归纳假设得到 $\Gamma\vdash t_1':T_{11}\to T_{12}$。把它和(也来自 T-App 情况的假设)$\Gamma\vdash t_2:T_{11}$ 结合起来,可以应用规则 T-App 得出 $\Gamma\vdash t':T$。

子情况 E-APP2: $t_1 = v_1$ (i.e., t_1 is a value)
$t_2 \to t_2'$
$t' = v_1\ t_2'$

类似。

子情况 E-APPABS: $t_1 = \lambda x:T_{11}.\ t_{12}$
$t_2 = v_2$
$t' = [x\mapsto v_2]t_{12}$

利用逆转引理,我们可以析构 $\lambda x:T_{11}.t_{12}$ 的类型化推导,生成 $\Gamma, x:T_{11} \vdash t_{12}:T_{12}$。由此根据代换引理(9.3.8),得到 $\Gamma \vdash t':T_{12}$。

这些情况对布尔常数和条件表达式与在定理(8.3.3)中的是相同的。

9.3.10 解答: 项 $(\lambda x:Bool.\lambda y:Bool)(true\ true)$ 是不良类型,但可归约为良类型的 $(\lambda y:Bool.y)$。

9.4.1 解答: T-True 和 T-False 是引入规则。T-If 是消去规则。T-Zero 和 T-Succ 是引入规则。T-Pred 和 T-Iszero 是消去规则。决定 succ 和 pred 是引入还是消去形式要求一些思考,因为它们既可创造数字又可使用数字。关键在于,当 pred 和 succ 相遇时,它们产生一个约式。对 iszero 是类似的。

11.2.1 解答:

```
t1    =  (λx:Unit.x) unit
ti+1  =  (λf:Unit→Unit. f(f(unit))) (λx:Unit.ti)
```

11.3.2 解答:

$$\frac{\Gamma \vdash t_2 : T_2}{\Gamma \vdash \lambda_:T_1.t_2 : T_1 \to T_2} \quad \text{(T-Wldcard)}$$

$$(\lambda_:T_{11}.t_{12})\ v_2 \longrightarrow t_{12} \quad \text{(T-Wildcard)}$$

这些规则的证明是缩写中推导出的,与定理(11.3.1)十分相似。

11.4.1 解答: 第(1)部分容易:如果我们使用规则 t as T = (λx :T.x)t,x 化简归属,那么可直接检查出归属的类型和求值规则能直接从抽象和应用的规则中推导出。

如果我们归属的求值规则改为一个迫切的版本,那么需要一个更精炼的化简方式,它可以延迟 t 的求值,直到归属被抛掉。例如,可以使用:

$$t \text{ as } T \stackrel{\text{def}}{=} (\lambda x:Unit\to T.\ x\ unit)\ (\lambda y:Unit.\ t) \quad \text{where } y \text{ is fresh.}$$

当然,这里的 Unit 选择是不重要的:任何类型都可以。

这里的微妙之处是,尽管化简是正确的,但是没有给出定理(11.3.1)的确切性质。原因是高阶规则 E-Ascribe 一步执行,一个化简的归属需要两步来求值到消失。但是,这不会使我们吃惊:我们可以把这个化简当做一个简单编译形式,而且观察到几乎每个高阶构造的编译形式需要对源语言中的每个原子归约,在目标语言中进行多步求值。于是我们所要做的是降低定理(11.3.1)的要求,规定每个高阶求值步骤应该由一些低阶步骤序列来匹配:

$$\text{if } t \longrightarrow_E t', \text{ then } e(t) \longrightarrow^*_I e(t')$$

最后的微妙在于另一个方向。也就是说,一个化简项的归约总能被"回射"到原来项的归约——精确说明时需稍加注意,因为消除一个化简归属需要两步,而且在第一步之后低阶项不会对应对原始高阶归属项的化简或已经消去的项的化简形式。事实是,尽管化简项的第一次归约总是能通过多步到达另一高阶项的化简来"完成"。正式地,如果 $e(t) \to_I s$,那么 $s \to^* e(t')$,有 $t \mapsto_E t'$。

11.5.1　解答：这是你需要加的：

```
let rec eval1 ctx t = match t with
    ...
  | TmLet(fi,x,v1,t2) when isval ctx v1 →
      termSubstTop v1 t2
  | TmLet(fi,x,t1,t2) →
      let t1' = eval1 ctx t1 in
      TmLet(fi, x, t1', t2)
    ...
let rec typeof ctx t =
  match t with
    ...
  | TmLet(fi,x,t1,t2) →
      let tyT1 = typeof ctx t1 in
      let ctx' = addbinding ctx x (VarBind(tyT1)) in
      (typeof ctx' t2)
```

11.5.2　解答：这个定义作用不大。第一，它改变了求值的顺序：规则 E-LetV 和规则 E-Let 说明了一个按值调用的顺序，其中 let x = t_1 in t_2 中的 t_1 必须允许将它代换为 x，并作用于 t_2 之前归约为一个值。第二，尽管类型化规则 T-Let 的有效性仍被保持——这直接可从置换引理(9.3.8)得出(项的不良类型性质不被保持)。例如，不良定义类型项：

```
let x = unit(unit) in unit
```

解释成良定义类型项 unit：由于 x 没有出现在 unit 主体中，不良定义类型子项 unit(unit)就消失了。

11.8.2　解答：图 A.1 给出了一个建议：增加类型到记录模式。对于概化的 let 结构的类型化规则，T-Let，提到一个单独的"模式类型" 关系(从算法上看)采用了一种模式和类型作为输入，如果成功，返回一个模式中给变量一个适当的绑定的上下文。T-Let 在主体 t_2 的类型检查中将这个上下文添加到当前上下文 Γ 中(我们假定记录模式不同字段囿变量集合是不相交的)。

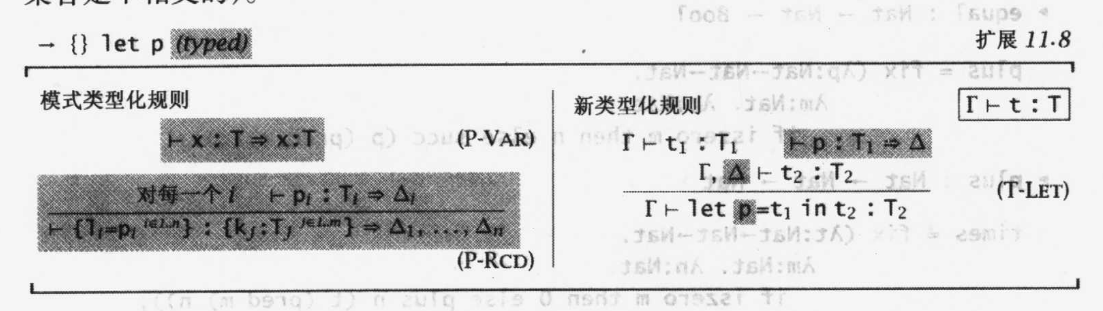


图 A.1　类型化的记录模式

注意只要我们愿意，稍微改进一下记录模式类型规则，让模式提到的字段比要匹配的值实际提供的字段少：

$$\frac{\{l_i{}^{i\in 1..n}\} \subseteq \{k_j{}^{j\in 1..m}\} \qquad \forall^{i\in 1..n}.\ \exists^{j\in 1..m}.\ l_i = k_j \text{ and } \vdash p_i : T_j \Rightarrow \Delta_i}{\vdash \{l_i{=}p_i{}^{i\in 1..n}\} \in \{k_j{:}T_j{}^{j\in 1..m}\} \Rightarrow \Delta_1, \ldots, \Delta_n} \qquad \text{(P-Rcd')}$$

如果采用这个规则,可将作为原语结构的记录投影形式除去,因为它现在能如语法修饰那样处理:

t.l $\stackrel{\text{def}}{=}$ let {l=x}=t in x

证明扩展系统的保持几乎和证明简单类型化 lambda 演算的保持一样。惟一需要扩充的是将模式类型关系与运行时的匹配操作联系起来的引理。如果 σ 是一个代换,且 Δ 是一个与 σ 有同样范围的上下文,那么 $\Gamma \vdash \sigma \models \Delta$ 表示对每一 $x \in dom(\Delta)$,我们有 $\Gamma \vdash \sigma(x):\Delta(x)$。

引理:如果 $\Gamma \vdash t:T$ 和 $\vdash p:T \Rightarrow \Delta$,那么 $match(p,t) = \sigma$,且 $\Gamma \vdash \sigma \models \Delta$。

这里提出的额外的符号还需要标准代换引理(9.3.8)对其进行稍微的概括。

引理:$\Gamma,\Delta \vdash t:T$ 和 $\Gamma \vdash \sigma \models \Delta$,那么 $\Gamma \vdash \sigma t:T$。

对于这里 C-Let 规则的保持证明可按以前同样的方式得出,只要根据 C-Let 情况运用这些引理。

11.9.1 解答:

Bool	$\stackrel{\text{def}}{=}$	Unit+Unit
true	$\stackrel{\text{def}}{=}$	inl unit
false	$\stackrel{\text{def}}{=}$	inr unit
if t_0 then t_1 else t_2	$\stackrel{\text{def}}{=}$	case t_0 of inl $x_1 \Rightarrow t_1$ | inr $x_2 \Rightarrow t_2$ where x_1 and x_2 are fresh.

11.11.1 解答:

```
equal =
  fix
    (λeq:Nat→Nat→Bool.
       λm:Nat. λn:Nat.
         if iszero m then iszero n
         else if iszero n then false
         else eq (pred m) (pred n));
▸ equal : Nat → Nat → Bool

plus = fix (λp:Nat→Nat→Nat.
             λm:Nat. λn:Nat.
               if iszero m then n else succ (p (pred m) n));
▸ plus : Nat → Nat → Nat

times = fix (λt:Nat→Nat→Nat.
              λm:Nat. λn:Nat.
                if iszero m then 0 else plus n (t (pred m) n));
▸ times : Nat → Nat → Nat

factorial = fix (λf:Nat→Nat.
                  λm:Nat.
                    if iszero m then 1 else times m (f (pred m)));
▸ factorial : Nat → Nat

factorial 5;
▸ 120 : Nat
```

11.11.2 解答:

```
letrec plus : Nat→Nat→Nat =
  λm:Nat. λn:Nat.
    if iszero m then n else succ (plus (pred m) n) in
letrec times : Nat→Nat→Nat =
  λm:Nat. λn:Nat.
    if iszero m then 0 else plus n (times (pred m) n) in
letrec factorial : Nat→Nat =
  λm:Nat.
    if iszero m then 1 else times m (factorial (pred m)) in
factorial 5;
▸ 120 : Nat
```

11.12.1 解答:奇怪！实际上,进展定理不支持。例如,表达式 head[T]nil[T]受阻——没有求值规则应用于它,但它不是值。对于一个成熟的程序语言,这种状况应由 head[T] nil[T]提升一个异常而不是受阻来处理:在第 14 章考虑异常。

11.12.2 解答:不能全部删除:如果我们抹除了 nil 的注释,那么就失去了类型的惟一性定理。操作上,当类型检查器看到 nil 就知道必须对 T 赋予类型 List T,但是不采用一些猜测形式,它就不知道如何选择 T。当然,更复杂的类型算法如 OCaml 编译器使用能精确地进行这种猜测。在第 22 章谈到这一点。

12.1.1 解答:对应用情况进行分析。假设试图说明 $t_1 t_2$ 是规范的。我们知道根据归纳假设 t_1 和 t_2 都是规范的,设 v_1 和 v_2 为它们的范式。根据类型关系引理(9.3.1)的逆转引理,我们知道 v_1 对某 T_{11} 和 T_{12} 有类型 $T_{11} \to T_{12}$。所以,根据典型形式引理(9.3.4),v_1 必有形式 $\lambda x:T_{11}.t_{12}$。但是归约 $t_1 t_2$ 为$(\lambda x:T_{11}.t_{12})v_2$ 没给出一个范式,由于这里应用E-AppAbs 规则,生成$[x \mapsto v_2]t_{12}$,要完成证明,必须证明该项是可规范化的。但这里我们受阻了,因为这一项可能(大体上)比原始项 $t_1 t_2$ 大(因为代换造成的 v_2 的副本与 t_{12} 中 x 出现的次数一样多)。

12.1.7 解答:定义(12.1.2)可用两个附加语句来扩展:

- R_{Bool}(t)当且仅当 t 停止
- $R_{T_1 \times T_2}$(t)当且仅当 t 停止,R_{T_1}(t.1),且 R_{T_2}(t.2)

引理(12.1.4)的证明可用一个附加的情况来扩展:

- 假定对于某 T_1 和 T_2,有 $T = T_1 \times T_2$。对于“仅当”方向(⇒)我们假定 R_T(t)且必须说明 R_T(t′),其中 t →t′。我们知道 R_{T_1} t.1 和 R_{T_2}(t.2)来自 $R_{T_1 \times T_2}$ 的定义。但是求值规则 E-Proj1 和 E-Proj2 告诉我们 t.1 → t′.1 和 t.2→ t′.2,所以根据归纳假设,有 R_{T_1}(t′.1) 和 R_{T_2}(t′.2)。据此,$R_{T_1 \times T_2}$ 的定义产生 $R_{T_1 \times T_2}$(t′)。“当且”方向是类似的。

最后,我们为引理(12.1.5)的证明需要补充一些情况(每一种情况对应着一条新的类型规则):

情况 T-IF: $\;\;$ `t = if t₁ then t₂ else t₃` $\;\Gamma \vdash t_1 : \mathtt{Bool}$
$\Gamma \vdash t_2 : T \qquad \Gamma \vdash t_3 : T$
where $\Gamma = x_1{:}T_1, \ldots, x_n{:}T_n$

设 $\sigma = [x_1 \mapsto v_1]\cdots[x_n \mapsto v_n]$。根据归纳假设,我们有 $R_{\mathrm{Bool}}(\sigma t_1)$,$R_T(\sigma t_2)$和 $R_T(\sigma t_3)$。根据引理(12.1.3),σt_1,σt_2 和 σt_3 都是规范的;让我们将它们的值写为 v_1,v_2 和 v_3。引理(12.1.4)给出 $R_{\mathrm{Bool}}(v_1)$,$R_T(v_2)$和 $R_T(v_3)$。同样,σt 自身明显是规范的。

我们对 T 继续归纳:

- 如果 T = A 或 T = Bool,那么 $R_T(\sigma t)$直接根据 σt 是规范的事实得出。
- 如果 $T = T_1 \to T_2$,那么必须说明任意的 $s \in R_{T_1}$ 有 $R_{T_2}\,\sigma t\ s$。所以假设 $s \in R_{T_1}$。接着,根据 if 表达式的求值规则,我们知道或者 $\sigma t\ s \longrightarrow^* v_2\ s$ 成立,或者 $\sigma t\ s \longrightarrow^* v_3\ s$ 成立,全依赖 v_1 是 true 还是 false。但我们由 $R_{T_1 \to T_2}$ 的定义知道 $R_{T_2}(v_2\ s)$,$R_{T_2}(v_3\ s)$还有 $R_{T_1 \to T_2}\,v_2$ 和 $R_{T_1 \to T_2}\,v_3$ 的事实。通过引理(12.1.4),有 $R_{T_1 \to T_2}(\sigma t\ s)$。

情况 T-TRUE: $\;\;$ `t = true` $\quad$ `T = Bool`

直接可证。

情况 T-FALSE: $\;\;$ `t = false` $\quad$ `T = Bool`

直接可证。

情况 T-PAIR: $\;\;$ $t = \{t_1, t_2\} \quad \Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2 \quad T = T_1 \times T_2$

根据归纳假设,对于 i = 1,2 有 $R_{T_i}(\sigma t_i)$。令 v_i 为对每个 σt_i 的范式。注意,由引理(12.1.4)有 $R_{T_i}(v_i)$。

对第一投影和第二投影应用求值规则有 $\{\sigma t_1, \sigma t_2\}.i \longrightarrow^* v_i$,由此根据引理(12.1.4),得到 $R_{T_i}(\{\sigma t_1, \sigma t_2\}.i)$。$R_{T_1 \times T_2}$ 的定义生成 $R_{T_1 \times T_2}(\{\sigma t_1, \sigma t_2\})$,即为 $R_{T_1 \times T_2}(\sigma\{t_1, t_2\})$。

情况 T-PROJ1: $\;\;$ $t = t_0.1 \quad \Gamma \vdash t_0 : T_1 \times T_2 \quad T = T_1$

直接从 $R_{T_1 \times T_2}$ 的定义得到。

13.1.1 解答:

```
a = {/ , \}          b = { \ , / }
    ↓     ↓               ↓ ↓
   (0)   (0)              (0)
```

13.1.2 解答:否,用任意索引号来调用 lookup 而不是 update,这将会造成发散。关键在于在用新的函数重写 a 之前要保证能找到 a 中存放的值,否则当我们开始查找时,只能找到新函数,而找不到原来的函数。

13.1.3 解答:假设语言提供了一个原语 free,它将引用单元作为参数,释放它的存储以便引用单元的下一次分配能得到同样的内存单元。那么程序:

```
let r = ref 0 in
let s = r in
free r;
let t = ref true in
t := false;
succ (!s)
```

将求值为受阻项 succ false。注意别名在出现的问题中起到了一个关键的作用,因为如果变量 r 自由出现在表达式的其余部分,它就会产生非法的 free r,这样会妨碍我们检测无效的回收。

这种例子在类似 C 这样的手工内存分配管理的语言中容易模仿(ref 构造子对应于 C 的 malloc,这里的 free 是对 C 中 free 的简化版)。

13.3.1　解答:一个简单的垃圾收集计算应为这样:

1. 通过使位置集$\mathcal{L}$有限来模仿内存的有限性。
2. 在存储中定义位置的可达性。将 t 所在的位置集写为 *locations*(t)。对位置 l, l' 和存储 μ,如果 $l' \in locations(\mu(l))$,那么在 μ 中,l 可一步到达 l'。如果存在位置的有限序列开始于 l 终止于 l',那么 l' 是 l 可以到达的,而且每一步都是可以从前一步到达的。最后,定义可以从存储 μ 中的项 t 为可达的位置集,记为 *reachable*(t, μ),作为 μ 中从 *locations*(t)可达的位置集合。
3. 用关系 $t|\mu \rightarrow_{gc} t|\mu'$ 模拟垃圾收集行为,其中该关系由以下规则来定义:

$$\frac{\mu' = \mu \text{ 限制到 } (t, \mu)}{t \mid \mu \longrightarrow_{gc} t \mid \mu'} \qquad \text{(E-GC)}$$

   [μ' 的定义域就是 *reachable*(t, μ),在这个域的每个位置的值都和 μ 的一样]。

4. 将求值序列定义为求值与垃圾收集交错进行的序列:$\xrightarrow{gc}{}^{*} \stackrel{\text{def}}{=} (\rightarrow \cup \rightarrow_{gc})^{*}$,注意我们不是仅加入规则 GC 到普通一步求值关系中,仅在"最外层"执行垃圾回收是很重要的,这里我们能看到整个项都被求值。如果允许垃圾收集发生在一个应用的左端求值"内部",例如,我们可能错误的重复使用了在应用中右端实际用到的位置,因为通过查看左端,我们看不出它们仍然为可存取的。
5. 还需要判断一下对估值规则定义的修改除了会消耗内存外,不会对最终结果产生影响:

   (a) 如果 $t|\mu \xrightarrow{gc}{}^{*} t'|\mu''$,那么 $t|\mu \longrightarrow^{*} t'|\mu'$,对这样一些 μ' 有 μ' 有比 μ'' 大的域,在它们被定义处一致。

   (b) 如果 $t|\mu \xrightarrow{gc}{}^{*} t'|\mu'$,那么或者:

      i. $t|\mu \xrightarrow{gc}{}^{*} t'|\mu''$,对一些 μ'',μ'' 有比 μ' 小的域,且在 μ' 被定义处与 μ' 一致
      ii. $t|\mu$ 的求值溢出内存,即到达状态 $t'''|\mu'''$,当求值的 t''' 下一步需要分配一个新空间,但由于 $reachable(t''', \mu''') = \mathcal{L}$ 而没有空间可用

这个简单的垃圾收集处理忽略了真实存储的几个方面,例如存储不同值分别要求不同长度的存储空间,像一些更高级语言特性那样,例如终结器(一段代码,运行系统要将垃圾回

收附在这段代码上的数据结构时执行它)和弱指针(不作为一个数据结构的"真实引用"的指针,因此该数据结构能够被垃圾收集,即使指向它的弱指针仍然存在)。在操作语义中一个更复杂的垃圾收集处理能在 Morrisett 等(1995)中找到。

13.4.1 解答:

```
let r₁ = ref (λx:Nat.0) in
let r₂ = ref (λx:Nat.(!r₁)x) in
(r₁ := (λx:Nat.(!r₂)x);
 r₂);
```

13.5.2 解答:让 μ 成为具有单独位置 l 的存储:

$$\mu \quad = \quad (l \mapsto \lambda x{:}\mathtt{Unit}.\ (!l)(x)),$$

Γ 为空上下文。根据以下两种存储类型 μ 是良类型的:

$$\Sigma_1 \quad = \quad l{:}\mathtt{Unit}\to\mathtt{Unit}$$
$$\Sigma_2 \quad = \quad l{:}\mathtt{Unit}\to(\mathtt{Unit}\to\mathtt{Unit}).$$

13.5.8 解答:在非强规范化的系统中存在良定义类型项。练习 13.1.2 给出了一个。这是另一个:

```
t1 = λr:Ref (Unit→Unit).
        (r := (λx:Unit. (!r)x);
         (!r) unit);
t2 = ref (λx:Unit. x);
```

将 t1 应用到 t2 产生了一个(良定义类型)发散项。

更概括地,我们能利用以下几步通过使用引用来定义任意递归函数(这一技术实际上被应用于一些函数语言的实现)。

1. 分配一个 ref 单元而且用一个有适当类型的虚函数来初始化:

```
fact_ref = ref (λn:Nat.0);
```

```
▸ fact_ref : Ref (Nat→Nat)
```

2. 定义我们感兴趣的函数体,利用递归调用的引用单元的内容:

```
fact_body =
   λn:Nat.
      if iszero n then 1 else times n ((!fact_ref)(pred n));
```

```
▸ fact_body : Nat → Nat
```

3. 通过存储实体到引用单元"回填":

```
fact_ref := fact_body;
```

4. 抽出引用单元的内容并按照要求使用:

```
fact = !fact_ref;
fact 5;
```

```
▸ 120 : Nat
```

14.1.1 解答:用 error 想要的类型注释它将会破坏类型保持性质。例如,一个良定义类型项:

```
(λx:Nat.x) ((λy:Bool.5) (error as Bool));
```

(这里 error as T 是异常的类型注释语法)将一步求值到一个不良定义类型项：

```
(λx:Nat.x) (error as Bool);
```

当求值规则将一个 error 从它出现的那一点传送至程序的最高层，我们可以将它视为有不同类型(T-Error 规则具有灵活性允许我们这样做)。

14.3.1　解答：标准 ML 的定义(Milner, Tofte, Harper 和 MacQueen, 1997; Milner 和 Tofte, 1991b)形式化了 exn 类型。一个相关的处理可以在 Harper 和 Stone(2000)中被找到。

14.3.2　解答：参看 Leroy 和 Pessaux(2000)。

14.3.3　解答：参看 Harper 等(1993)。

15.2.1　解答：

```
------------------------------ S-RcdPerm   -------------------------- S-RcdWidth
   {x:Nat,y:Nat,z:Nat}                     {y:Nat,x:Nat,z:Nat}
<: {y:Nat,x:Nat,z:Nat}                         <: {y:Nat}
---------------------------------------------------------------------- S-Trans
                 {x:Nat,y:Nat,z:Nat} <: {y:Nat}
```

15.2.2　解答：有很多同样结论的其他推导。这是一个：

```
                 ---------- S-Rcd-Width  ---------- S-Refl
      :          Rxy <: Rx               Nat <: Nat
---------------  ---------------------------------- S-Arrow      :
⊢ f : Rx→Nat              Rx→Nat <: Rxy→Nat                 -----------
----------------------------------------------- T-Sub       ⊢ xy : Rxy
              ⊢ f : Rxy→Nat
---------------------------------------------------------------------- T-App
                              ⊢ f xy : Nat
```

这是另一个：

```
                                  ---------- S-RcdWidth  -------- S-Refl
                       :          Rxy <: Rx              Rx <: Rx
      :           ------------ T-Rcd ---------------------------- S-Trans
---------------   ⊢ xy : Rxy               Rxy <: Rx
⊢ f : Rx→Nat      ------------------------------------------ T-Sub
                                ⊢ xy : Rx
------------------------------------------------ T-App
                ⊢ f xy : Nat
```

实际上，如第二个例子建议的，在这个演算中对任何可推导的语句实际上都有无限多类型化推导。

15.2.3　解答：

1. 有 6 个：{a:Top, b:Top}，{b:Top, a:Top}，{a:Top}，{b:Top}，{}和 Top。
2. 例如，令：

   $S_0 = \{\}$

   $S_1 = \{a:Top\}$

   $S_2 = \{a:Top, b:Top\}$

   $S_2 = \{a:Top, b:Top, c:Top\}$

   等等。
3. 例如，令 $T_0 = S_0 \to Top$，$T_1 = S_1 \to Top$ $T_2 = S_2 \to Top$，等等。

15.2.4 解答:(1)否。如果存在一个,它必将或者是一个箭头类型或者是一个记录类型(显然它不可能是 Top)。但是一个记录类型不可能是任何箭头类型的子类型,反之亦然。(2)再次否。如果有这样的箭头类型 $T_1 \to T_2$,它的定义域类型 T_1 将是每一其他类型 S_1 的子类型;但我们刚看到这是不可能的。

15.2.5 解答:如果我们想要保持存在的求值语义,增加这个规则将不是好方法。新的规则允许我们去推论,例如:Nat × Bool <: Nat,这将表明受阻项(succ(5,true))将是良定义类型,违背了进展定理。这个规则在一个"强迫语义"中是安全的(参见 15.6 节),尽管对子类型的检查会产生算法困难。

15.3.1 解答:通过增加伪造序对到子类型关系,我们能够打破保持和进展。例如,增加公理:

$$\{x:\{\}\} <: \{x:Top \to Top\}$$

允许我们推导 $\Gamma \vdash t:Top$,其中 $t = (\{x = \{\}\}.x)\{\}$。但是 $t \to \{\}\{\}$,是不可类型化的——违反了保持规则。或者,若增加公理:

$$\{\} <: Top \to Top$$

于是{}{}项是可类型化的,但这一项受阻而不是值——违反进展定理。

另一方面,从子类型关系中取出序对是没有危险的。在进展性质陈述中提到类型关系的惟一位置是在前提中,所以限制子类型关系,进而类型关系只能使进展性质更容易获得。关于保持性质的情况,特别地,我们担心抛弃传递性规则将引起麻烦。直觉上,不能的原因是由于它在系统中的作用实际上不是关键的,因为包含规则 T-Sub 可以替代传递性规则,例如,不用:

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash t \in S} \qquad \dfrac{\dfrac{\vdots}{S <: U} \quad \dfrac{\vdots}{U <: T}}{S <: T}\ \text{S-TRANS}}{\Gamma \vdash t : T}\ \text{T-SUB}$$

可写为:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash t : S} \quad \dfrac{\vdots}{S <: U}}{\Gamma \vdash t \in U}\ \text{T-SUB} \qquad \dfrac{\vdots}{U <: T}}{\Gamma \vdash t : T}\ \text{T-SUB}$$

15.3.2 解答:

1. 根据归纳于子类型化推导。通过检查图 15.1 和图 15.3 的子类型化规则,在推导 $S <: T_1 \to T_2$ 中得最终规则显然是 S-Refl,S-Trans 或 S-Arrow。

   如果最终规则是 S-Refl,那么结果可直接得出(由于在这种情况中 $S = T_1 \to T_2$,我们可以通过自反性推导出 $T_1 <: T_1$ 和 $T_2 <: T_2$)。

   如果最终规则是 S-Trans,那么对一些类型 U 我们有结论为 $S <: U$ 和 $U <: T_1 \to T_2$ 的子推

导,通过对第二个子推导归纳假设,我们看到 U 有形式 $U_1 \to U_2$ 其中 T_1 <: U_2 和 U_2 <: T_2。现在,既然我们知道 U 是一个箭头类型,我们可以应用归纳假设到第一个子推导来得到满足 U_1 <: S_1 和 S_2 <: U_2 下的 $S = S_1 \to S_2$。最终,可以使用 S-Trans 两次来重新组织我们已经建立的事实,得到 T_1 <: S_1(从 T_1 <: U_1 和 U_1 <: S_1)和 S_2 <: T_2(从 S_2 <: U_2 和 U_2 <: T_2)。

如果最终规则是 S-Arrow,那么 S 有理想的形式且直接子推导正是我们所需要的关于 S 部分的事实。

2. 根据归纳于子类型推导。再次检查子类型规则,揭示 S <: $\{l_i:T_i^{i\in l..n}\}$ 中的推导最终规则必定是 S-Refl, S-Trans, S-RcdWidth, S-RcdDepth, 或者 S-RcdPerm。对S-Refl 的情况是平凡的。对 S-RcdWidth, S-RcdDepth 和 S-RcdPerm 的情况都直接可证。

   如果最终规则是 S-Trans,那么我们对一些类型 U 有结论为 S <: U 和 U <: $\{l_i:T_i^{i\in l..n}\}$ 的子推导。应用归纳假设到第二个子推导,我们看到 U 对于每个 $l_i = u_a$ 有形式 $\{u_a: U_a^{a\in 1..0}\}$,其中 $\{l_i^{i\in 1..n}\} \subseteq \{u_a^{a\in 1..0}\}$ 和 U_a <: T_i。现在既然知道 U 是一个记录类型,我们能够应用归纳假设到第一个子推导来得到对于每个 $u_a = k_j$ 有 $S = \{k_j:S_j^{j\in 1..m}\}$,其中 $\{u_a^{a\in 1..0}\} \subseteq \{k_j^{j\in 1..m}\}$ 和 S_j <: U_a。通过重新整理事实,可根据集合包含的传递性得到 $\{l_i^{i\in 1..n}\} \subseteq \{k_j^{j\in 1..m}\}$ 和通过 S-Trans 对于每一个 $l_i = k_j$ 有 S_j <: T_i,由于每一个 T 中的标签必须同时在 U 中(或者说 l_i 对某些 a 来说必须等于 u_a),以及接下来我们知道 S_j <: U_a 和 U_a <: T_i(这里关于元变量名的艰难选择是无法避免的:这里没有足够的罗马字符随处可用)。

15.3.6 解答:两部分都可根据对类型化推导进行归纳。我们仅显示第一部分的证明。
通过对类型化规则的检查,在推导⊢ $v:T_1 \to T_2$ 中的最终规则显然是 T-Abs 或T-Sub。如果是 T-Abs 那么我们希望的结果直接来自规则的前提,所以假设最终规则是 T-Sub。
根据 T-Sub 的前提,有⊢ $v:S$ 和 S <: $T_1 \to T_2$。从逆转引理(15.3.2),知道 S 有形式 $S_1 \to S_2$。结果可以从归纳假设中推出。

16.1.2 解答:第(1)部分是对结构 S 的直接归纳。
对于第(2)部分,根据第(1)部分首先注意到,如果存在关于 S <: T 的任何推导,那么必存在一个无自反的推导。我们现在根据归纳于 S <: T 的无自反的推导长度继续讨论。注意到我们对推导的长度来归纳,而不是像我们过去那样只对它的结构。这是必要的,因为在箭头和记录情况中,我们对未以原始的子推导形式出现的新构造的推导进行归纳。
如果推导中的最后规则为除了 S-Trans 外的任何规则,那么结果可以从归纳假设中直接得出(换句话说,通过归纳假设,所有最终规则的子推导能被不包括传递性的推导替换;因为最终规则自身也不是可传递的,整个推导现在都是无传递性的)。所以假设最终规则是 S-Trans。换句话说,我们得到了对一些类型 U 含结论 S <: U 和 U <: T 的子推导。接下来考虑在这两种子推导中的最后规则序对的情况:

情况 ANY/S-TOP: T = Top

如果右端推导结束于 S-Top，那么结果是直接的，由于 S <: Top 能从 S-Top 推导出，无论 S 是什么。

情况 S-TOP/ANY: $U = Top$

如果左端推导结束于 S-Top，我们首先注意到，通过推导假设，可以假定右端子推导是无传递性的。现在，通过检查子类型化规则，可看到这个子推导的最后规则一定是 S-Top（我们已经取消自反性，而且所有其他规则要求 U 或是箭头或是记录）。结果通过 S-Top 得出。

情况 S-ARROW/S-ARROW: $S = S_1 \to S_2$ $U = U_1 \to U_2$ $T = T_1 \to T_2$
$U_1 <: S_1$ $S_2 <: U_2$
$T_1 <: U_1$ $U_2 <: T_2$

使用 S-Trans 可以从给出的子推导构造 $T_1 <: S_1$ 和 $S_2 <: T_2$ 的推导。此外，这些新的推导是严格地小于原来的推导，所以归纳假设能被应用以得出无传递的 $T_1 <: S_1$ 和 $S_2 <: T_2$ 的推导。把这些和 S-Arrow 结合，可得到无传递的 $S_1 \to S_2 <: T_1 \to T_2$ 的推导。

情况 S-Rcd/S-Rcd：

类似。

其他情况：

其他组合方式（S-Arrow/S-Rcd 和 S-Rcd/S-Arrow）是不可能的，由于它们在 U 的形式上加上了不相容的约束。

16.1.3 解答：如果我们增加 Bool 类型，那么引理（16.1.2）的第一部分需要被修改一下，它现在应该读做“不使用 S-Refl，S <: S 能对任何类型 S 都是可推导的，除 Bool 类型外”或者另一种方法，我们可以增加一个规则：

```
Bool <: Bool
```

到子类型定义中，而接着显示 S-Refl 能从扩展系统删除。

16.2.5 解答：根据对声明性类型化推导进行归纳。然后分析推导中的最终规则的情况。

情况 T-VAR: $t = x$ $\Gamma(x) = T$

通过 TA-Var 直接得出。

情况 T-ABS: $t = \lambda x{:}T_1.t_2$ $\Gamma, x{:}T_1 \vdash t_2 : T_2$ $T = T_1 \to T_2$

通过归纳假设，对一些 $S_2 <: T_2$ 有 $\Gamma, x : T_1 \mapsto t_2 : S_2$。由 TA-Abs 有 $\Gamma \mapsto t : T_1 \to S_2$。由 S-Arrow 有 $T_1 \to S_2 <: T_1 \to T_2$，如要求的那样。

情况 T-APP: $t = t_1\ t_2$ $\Gamma \vdash t_1 : T_{11} \to T_{12}$ $\Gamma \vdash t_2 : T_{11}$ $T = T_{12}$

通过归纳假设，对于一些 $S_1 <: T_{11} \to T_{12}$ 有 $\Gamma \mapsto t_1 : S_1$ 且对于一些 $S_2 <: T_{11}$ 有 $\Gamma \mapsto t_2 : S_2$。通过子类型关系的逆转引理（15.3.2），对一些 S_{11} 和 S_{12} 且 $T_{11} <: S_{11}$ 和 $S_{12} <: T_{12}$，必有形式 $S_{11} \to S_{12}$。根据传递性，有 $S_2 <: S_{11}$。根据算法子类型的完备性，有 $\mapsto S_2 <: S_{11}$。现在由 TA-App，$\Gamma \mapsto t_1 : t_2 : S_{12}$，这个情况讨论结束（因为我们已有 $S_{12} <: T_{12}$）。

情况 T-RCD: $t = \{l_i{=}t_i\ ^{i\in 1..n}\}$ $\quad \Gamma \vdash t_i : T_i$ 对每个 i
$T = \{l_i{:}T_i\ ^{i\in 1..n}\}$

直接可证。

情况 T-PROJ: $t = t_1.l_j \quad \Gamma \vdash t_1 : \{l_i{:}T_i\ ^{i\in 1..n}\} \quad T = T_j$

与应用情况类似。

情况 T-SUB: $t : S \quad S <: T$

根据归纳假设和子类型化传递性。

16.2.6 解答:根据声明性规则,项 λx: {a: Nat} .x 既有类型{a: Nat}→{a : Nat}又有{a :Nat}→Top。但没有 S-Arrow,这些类型就无法比较(而且没有存在于它们两者之下的类型)。

16.3.2 解答:先给出算法的互递归序对声明(而在下面将会显示),分别计算类型 S 和 T 序对的合类型 J 和交类型 M。第二个算法仍将会失败,表示为 S 和 T 没有交。

$$
S \vee T = \begin{cases}
\text{Bool} & \text{if } S = T = \text{Bool} \\
M_1 \to J_2 & \text{if } S = S_1 \to S_2 \quad T = T_1 \to T_2 \\
 & \quad S_1 \wedge T_1 = M_1 \quad S_2 \vee T_2 = J_2 \\
\{j_l{:}J_l\ ^{l\in 1..q}\} & \text{if } S = \{k_j{:}S_j\ ^{j\in 1..m}\} \\
 & \quad T = \{l_i{:}T_i\ ^{i\in 1..n}\} \\
 & \quad \{j_l\ ^{l\in 1..q}\} = \{k_j\ ^{j\in 1..m}\} \cap \{l_i\ ^{i\in 1..n}\} \\
 & \quad S_j \vee T_i = J_l \quad \text{对每个 } j_l = k_j = l_i \\
\text{Top} & \text{其他情况}
\end{cases}
$$

$$
S \wedge T = \begin{cases}
S & \text{if } T = \text{Top} \\
T & \text{if } S = \text{Top} \\
\text{Bool} & \text{if } S = T = \text{Bool} \\
J_1 \to M_2 & \text{if } S = S_1 \to S_2 \quad T = T_1 \to T_2 \\
 & \quad S_1 \vee T_1 = J_1 \quad S_2 \wedge T_2 = M_2 \\
\{m_l{:}M_l\ ^{l\in 1..q}\} & \text{if } S = \{k_j{:}S_j\ ^{j\in 1..m}\} \\
 & \quad T = \{l_i{:}T_i\ ^{i\in 1..n}\} \\
 & \quad \{m_l\ ^{l\in 1..q}\} = \{k_j\ ^{j\in 1..m}\} \cup \{l_i\ ^{i\in 1..n}\} \\
 & \quad S_j \wedge T_i = M_l \quad \text{对每个 } m_l = k_j = l_i \\
 & \quad M_l = S_j \qquad \text{if } m_l = k_j \text{ occurs only in S} \\
 & \quad M_l = T_i \qquad \text{if } m_l = l_i \text{ occurs only in T} \\
\textit{fail} & \text{其他情况}
\end{cases}
$$

在第一个算法的箭头情况中,对第二个算法的调用会导致失败;这种情况中第一个算法转变为最后一种情况且结果为 Top。例如 Bool → Top ∨ {} → Top = Top。
很容易检查出∨ 和∧ 是全函数(换句话说:从不发散)。注意,S 和 T 的中总长度在递归调用中总是减少。而且∧ 算法,当它的输入被如下界定时不会失败:

A.10 引理:对一些 M,如果 L <:S 和 L <:T,那么 S ∧ T = M。

证明:根据对 S(或等价的 T)的长度进行归纳,结合对 S 和 T 的形状进行分析的情况。如果 S 或 T 是 Top,那么定义中的头两个中的一个可应用,且结果分别是 T 或者 S。S 和 T 有

不同形状的情况不可能发生，因为子类型关系的逆转引理(15.3.2)将会在 L 形状上做出不一致的要求；例如，如果 S 是一个箭头类型，那么 L 一定是，但如果 T 是一个记录，那么 L 也是[①]。所以我们剩下三种情况：

如果 S 和 T 都是 Bool，那么定义中的第三种情况可应用，则证明结束。

假设 $S=S_1\to S_2$ 和 $T=T_1\to T_2$。$\vee$ 算法完整性告诉我们第一个递归调用返回某类型 J_1。同样通过逆转引理，L 必须有形式 $L_1\to L_2$ 且 L_2 <: S_2 和 L_2 <: T_2。也就是，L_2 是 S_2 和 T_2 的公共下界，所以可用归纳假设来告诉我们 $S_2\wedge L_2$ 没有失败，反而返回一个类型 M_2。所以 $S\wedge T=J_1\to M_2$。

最后，假设 $S=\{k_j:S_j^{j\in 1..m}\}$ 和 $T=\{l_i:T_i^{i\in 1..n}\}$。通过逆转引理，L 必须是一个其标签包括所有出现在 S 和 T 的标签的记录类型。而且，对 S 和 T 中的每一个标签，逆转引理告诉我们 L 中的对应字段是 S 和 T 字段的共有子类型。这给我们保证对有共有标签的 $\wedge$ 算法的递归调用都能成功。

现在让我们证实这些定义计算合类型和交类型。这个讨论被分为两部分：命题(A.11)显示这个交类型计算的是 S 和 T 的下界而合类型是上界；命题(A.12)显示被计算的交类型比 S 和 T 的每个公共下界都大，而合类型则小于每个公共上界。

A.11 命题：

1. 如果 $S\vee T=J$，那么 S <: J 和 T <: J。
2. 如果 $S\wedge T=M$，那么 M <: S 和 M <: T。

证明： 直接归纳于 $S\wedge T=M$ 或 $S\vee T=J$ 的"推导"长度(即对计算 M 或 J 而定义的 $\wedge$ 和 $\vee$ 的递归调用次数)。

A.12 命题：

1. 假设 $S\vee T=J$ 且，对某 U，有 S <: U 和 T <: U，那么 J <: U。
2. 假设 $S\wedge T=M$ 且，对某 L，有 L <: S 和 L <: T，那么 L <: M。

证明： 根据归纳于 S 和 T 的长度，这两部分一起证明(实际上，对任何东西归纳都可以)。给定 S 和 T，我们依次考虑这两部分。

对于第(1)部分，考虑形式 U 的情况。如果 U 是 Top，那么证明完成，因为无论 J 是什么，有 J <: Top。如果 U = Bool，那么逆转引理(15.3.2)告诉我们 S 和 T 必定也是 Bool，那么 J = Bool 证毕。其他的情况更有趣。

如果 $U=U_1\to U_2$，那么，通过逆转引理，$S=S_1\to S_2$ 和 $T=T_1\to T_2$，其中 U_1 <: S_1，U_1 <: T_1，S_2 <: U_2，和 T_2 <: U_2。根据归纳假设，S_1 和 T_1 的交类型 M_1 高于 U_1，S_2 和 T_2 的合类型 J_2，低于 U_2。由 S-Arrow，有 $M_1\to J_2$ <: $U_1\to U_2$。

如果 U 是一个记录类型，那么，通过逆转引理，S 和 T 也如此。而且，S 和 T 的标签是 U 标签的超集，且 U 中每个字段的类型是 S 和 T 的对应字段的超集。由此，S 和 T 的合类型将

① 严格地说，引理(15.3.2)无法解决 Bool。对 Bool 的增加情况仅说明 Bool 的惟一子类型是 Bool 本身。

至少包括 U 的标签，而且(通过归纳假设)合类型的字段将是的对应的 U 的子类型。由 S-Rcd，有 J <: U。

对于第(2)部分，我们再次讨论 S 和 T 的形式。如果一个是 Top，那么交类型是另一个，而且结果直接可证。S 和 T 有不同(非 Top)形状的情况不可能发生，像我们在证明(A.10)中看到的。如果两者都是 Bool，那么结果又是直接可证的。剩余情况(S 和 T 都是箭头或者都是记录)更有趣。

如果 $S = S_1 \to S_2$ 和 $T = T_1 \to T_2$，那么根据逆转引理，必定有 $L = L_1 \to L_2$，其中 $S_1 <: L_1$，$T_1 <: L_1$，$L_2 <: S_2$ 和 $L_2 <: T_2$。通过归纳假设，S_1 和 T_1 的合类型 J_1 低于 L_1，同时 S_2 和 T_2 的交类型 M_2 高于 L_2。由 S-Arrow，有 $L_1 \to L_2 <: J_1 \to M_2$。

如果 S 和 T 是记录类型，那么根据逆转引理，L 也一样。更进一步，L 必定有 S 和 T 的所有标签(可能更多)，对应的字段肯定符合子类型关系。现在，对每个 S 和 T 的交类型 M 中的每个标签 m_l，有三种可能。如果 m_l 存在于 S 和 T 中，那么它在 M 中类型是它在 S 和 T 中类型的交类型，而且根据归纳假设，在 L 中的对应类型是在 M 中类型的子类型。另一方面，如果 m_l 只存在于 S，那么在 M 中的相应类型和在 S 中的一样，而且我们已经知道 L 中的类型小一些。m_l 只在 T 中出现的情况是类似的。

16.3.3　解答：这一项的最小类型是 Top：Bool 和{}的合类型。但是，这一项是可类型化的事实应该被视为语言的缺点，因为很难想像程序员真的想写这个表达式。毕竟，在类型 Top 的值上没有操作能被执行，所以在第一位计算几乎没有意义！对这一弱点有两个可能的回答。一个是简单地从系统中移走 Top，而且将 ∨ 作为一个部分的操作。另一个是保留 Top，但只要当它遇到最小类型为 Top 的项时，类型检查器产生一个警告。

16.3.4　解答：处理 Ref 类型是直接的。我们仅增加一个子句到交类型和合类型算法：

$$S \vee T = \begin{cases} \ldots & \ldots \\ \mathtt{Ref(T_1)} & \text{if } S = \mathtt{Ref(S_1)},\ T = \mathtt{Ref(T_1)},\ S_1 <: T_1, \text{ and } T_1 <: S_1 \\ \ldots & \ldots \end{cases}$$

$$S \wedge T = \begin{cases} \ldots & \ldots \\ \mathtt{Ref(T_1)} & \text{if } S = \mathtt{Ref(S_1)},\ T = \mathtt{Ref(T_1)},\ S_1 <: T_1, \text{ and } T_1 <: S_1 \\ \ldots & \ldots \end{cases}$$

然而，当用 Source 和 Sink 构造子来改进 Ref 时，会遭遇到一个主要困难：子类型关系不再有合类型(或交类型)！例如，类型 Ref{a:Nat，b:Bool}和 Ref{a:Nat }同时是 Source{a:Nat }和 Sink{a:Nat，b:Bool}的子类型，但这些类型没有公共下界。

有不同的方式来解释这个困难。可能最简单的是增加一个 Source 或 Sink，但不能二者同时加到系统中。对很多应用领域，这就够了。例如，对 18.12 节中的被修改的类的实现，仅需要 Source。另一方面，在具有通道类型的并发语言中(参见第 15.5 节)，我们倾向于 Sink，因为这将给定义一个服务器进程的能力，而且在它的存取通道上仅传递“发送能力”(接收能力仅服务器进程自身需要)。

仅有 Source 类型，合类型算法经过如下改进仍然保持完备(从上面看，我们仍然需要 Ref 语句；类似语句被加到交类型算法)：

$$S \vee T = \begin{cases} \ldots & \ldots \\ \mathtt{Source}(J_1) & \text{if } S = \mathtt{Ref}(S_1) \quad T = \mathtt{Ref}(T_1) \\ & \quad S_1 \vee T_1 = J_1 \\ \mathtt{Source}(J_1) & \text{if } S = \mathtt{Source}(S_1) \quad T = \mathtt{Source}(T_1) \\ & \quad S_1 \vee T_1 = J_1 \\ \mathtt{Source}(J_1) & \text{if } S = \mathtt{Ref}(S_1) \quad T = \mathtt{Source}(T_1) \\ & \quad S_1 \vee T_1 = J_1 \\ \mathtt{Source}(J_1) & \text{if } S = \mathtt{Source}(S_1) \quad T = \mathtt{Ref}(T_1) \\ & \quad S_1 \vee T_1 = J_1 \\ \ldots & \ldots \end{cases}$$

(Hennessy 和 Riely 1998 年建议的)一个不同的解决是改进 Ref 类型构造子,这样不用一个参数,而采用两个:Ref S T 的元素是能用于存放类型 S 的元素及检索类型 T 的元素的引用单元。新的 Ref 是在它第一个参数中的逆变式和它第二参数中的协变式。现在 Sink S 能被定义为一个 Ref S Top 的缩写,而 Source T 能被定义为 Ref Bot T 的缩写。

16.4.1 解答:对:

$$\frac{\Gamma \vdash t_1 : T_1 \qquad T_1 = \mathtt{Bot} \qquad \Gamma \vdash t_2 : T_2 \qquad \Gamma \vdash t_3 : T_3 \qquad T_2 \vee T_3 = T}{\Gamma \vdash \mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3 : T} \qquad \text{(TA-If)}$$

可替换为规则:

$$\frac{\Gamma \vdash t_1 : T_1 \qquad T_1 = \mathtt{Bot} \qquad \Gamma \vdash t_2 : T_2 \qquad \Gamma \vdash t_3 : T_3}{\Gamma \vdash \mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3 : \mathtt{Bot}} \qquad \text{(TA-If)}$$

且是安全的(由于 Bot 为空,t_1 的求值永远不可能产生一个常规结果),但这个规则将一些类型赋给项,因为这些项不能由声明性类型规则赋予类型;选择它将破坏定理(16.2.4)。

17.3.1 解答:这个解答仅需要练习 16.3.2 中的算法。

```
    let rec join tyS tyT =
      match (tyS,tyT) with
  (TyArr(tyS1,tyS2),TyArr(tyT1,tyT2)) →
    (try TyArr(meet tyS1 tyT1, join tyS2 tyT2)
      with Not_found → TyTop)
| (TyBool,TyBool) →
    TyBool
| (TyRecord(fS), TyRecord(fT)) →
    let labelsS = List.map (fun (li,_) → li) fS in
    let labelsT = List.map (fun (li,_) → li) fT in
    let commonLabels =
      List.find_all (fun l → List.mem l labelsT) labelsS in
    let commonFields =
      List.map (fun li →
                  let tySi = List.assoc li fS in
                  let tyTi = List.assoc li fT in
                  (li, join tySi tyTi))
                commonLabels in
    TyRecord(commonFields)
| _ →
    TyTop
```

```
and meet tyS tyT =
  match (tyS,tyT) with
    (TyArr(tyS1,tyS2),TyArr(tyT1,tyT2)) →
      TyArr(join tyS1 tyT1, meet tyS2 tyT2)
  | (TyBool,TyBool) →
      TyBool
  | (TyRecord(fS), TyRecord(fT)) →
      let labelsS = List.map (fun (li,_) → li) fS in
      let labelsT = List.map (fun (li,_) → li) fT in
      let allLabels =
        List.append
          labelsS
          (List.find_all
            (fun l → not (List.mem l labelsS)) labelsT) in
      let allFields =
        List.map (fun li →
                    if List.mem li allLabels then
                      let tySi = List.assoc li fS in
                      let tyTi = List.assoc li fT in
                      (li, meet tySi tyTi)
                    else if List.mem li labelsS then
                      (li, List.assoc li fS)
                    else
                      (li, List.assoc li fT))
                 allLabels in
      TyRecord(allFields)
  | _ →
      raise Not_found

let rec typeof ctx t =
  match t with
    ...
  | TmTrue(fi) →
      TyBool
  | TmFalse(fi) →
      TyBool
  | TmIf(fi,t1,t2,t3) →
      if subtype (typeof ctx t1) TyBool then
        join (typeof ctx t2) (typeof ctx t3)
      else error fi "guard of conditional not a boolean"
```

17.3.2　解答: 参见 rcssubbot 的实现。

18.6.1　解答:

```
DecCounter = {get:Unit→Nat, inc:Unit→Unit, reset:Unit→Unit,
              dec:Unit→Unit};

decCounterClass =
  λr:CounterRep.
    let super = resetCounterClass r in
      {get   = super.get,
       inc   = super.inc,
       reset = super.reset,
       dec   = λ_:Unit. r.x:=pred(!(r.x))};
```

18.7.1 解答:

```
BackupCounter2 = {get:Unit→Nat, inc:Unit→Unit,
                  reset:Unit→Unit, backup: Unit→Unit,
                  reset2:Unit→Unit, backup2: Unit→Unit};
BackupCounterRep2 = {x: Ref Nat, b: Ref Nat, b2: Ref Nat};

backupCounterClass2 =
  λr:BackupCounterRep2.
    let super = backupCounterClass r in
      {get = super.get, inc = super.inc,
       reset = super.reset, backup = super.backup,
       reset2 = λ_:Unit. r.x:=!(r.b2),
       backup2 = λ_:Unit. r.b2:=!(r.x)};
```

18.11.1 解答:

```
instrCounterClass =
  λr:InstrCounterRep.
    λself: Unit→InstrCounter.
      λ_:Unit.
        let super = setCounterClass r self unit in
          {get = λ_:Unit. (r.a:=succ(!(r.a)); super.get unit),
           set = λi:Nat. (r.a:=succ(!(r.a)); super.set i),
           inc = super.inc,
           accesses = λ_:Unit. !(r.a)};

ResetInstrCounter = {get:Unit→Nat, set:Nat→Unit,
                     inc:Unit→Unit, accesses:Unit→Nat,
                     reset:Unit→Unit};

resetInstrCounterClass =
  λr:InstrCounterRep.
    λself: Unit→ResetInstrCounter.
      λ_:Unit.
        let super = instrCounterClass r self unit in
          {get = super.get,
           set = super.set,
           inc = super.inc,
           accesses = super.accesses,
           reset = λ_:Unit. r.x:=0};

BackupInstrCounter = {get:Unit→Nat, set:Nat→Unit,
                      inc:Unit→Unit, accesses:Unit→Nat,
                      backup:Unit→Unit, reset:Unit→Unit};

BackupInstrCounterRep = {x: Ref Nat, a: Ref Nat, b: Ref Nat};

backupInstrCounterClass =
  λr:BackupInstrCounterRep.
    λself: Unit→BackupInstrCounter.
      λ_:Unit.
        let super = resetInstrCounterClass r self unit in
          {get = super.get,
```

```
                set = super.set,
                inc = super.inc,
                accesses = super.accesses,
                reset = λ_:Unit. r.x:=!(r.b),
                backup = λ_:Unit. r.b:=!(r.x)};

newBackupInstrCounter =
  λ_:Unit. let r = {x=ref 1, a=ref 0, b=ref 0} in
               fix (backupInstrCounterClass r) unit;
```

18.13.1 解答:一种恒等测试的方法是使用引用单元。我们以类型 Ref Nat 的实例变量 id 扩展我们对象的内部表示:

```
IdCounterRep = {x: Ref Nat, id: Ref (Ref Nat)};
```

而且一个 id 方法仅返回 id 字段:

```
IdCounter = {get:Unit→Nat, inc:Unit→Unit, id:Unit→(Ref Nat)};
idCounterClass =
  λr:IdCounterRep.
    {get = λ_:Unit. !(r.x),
     inc = λ_:Unit. r.x:=succ(!(r.x)),
     id  = λ_:Unit. !(r.id)};
```

现在,sameObject 函数采用带 id 方法的两个对象,并检查 id 方法返回引用是否相同。

```
sameObject =
  λa:{id:Unit→(Ref Nat)}. λb:{id:Unit→(Ref Nat)}.
    ((b.id unit) := 1;
     (a.id unit) := 0;
     iszero (!(b.id unit)));
```

这个技巧是使用别名来检查两个引用单元是否相同:我们确定第二个为非零,给第一个赋值零,检查第二个来查看它是否变为零。

19.4.1 解答:由于每个声明必须包括 extends 子句,而且由于这些子句不允许循环,从每个类的 extends 子句链必须最终以 Object 结束。

19.4.2 解答:一个明显的改进将把三个强制转换(casting)类型化规则联合为一个:

$$\frac{\Gamma \vdash t_0 : D}{\Gamma \vdash (C)t_0 : C} \qquad \text{(T-Cast)}$$

并且删除愚蠢转型。另一个可忽略构造子,因为它们什么都不做。

19.4.6 解答:

1. FJ 接口的形式是常规的。
2. 假定我们声明了以下接口:

```
interface A {}
interface B {}
interface C extends A,B {}
interface D extends A,B {}
```

于是 C 和 D 同时有 A 和 B 共同作为公共上界,但没有最小上界。

3. 不采用条件表达式的标准算法规则：

$$\frac{\Gamma \vdash t_1 : \text{boolean} \qquad \Gamma \vdash t_2 : E_2 \qquad \Gamma \vdash t_3 : E_3}{\Gamma \vdash t_1 \; ? \; t_2 : t_3 : E_2 \vee E_3}$$

Java 使用以下严格的规则：

$$\frac{\Gamma \vdash t_1 : \text{boolean} \qquad \Gamma \vdash t_2 : E_2 \qquad \Gamma \vdash t_3 : E_3 \qquad \Gamma \vdash E_2 <: E_3}{\Gamma \vdash t_1 \; ? \; t_2 : t_3 : E_3}$$

$$\frac{\Gamma \vdash t_1 : \text{boolean} \qquad \Gamma \vdash t_2 : E_2 \qquad \Gamma \vdash t_3 : E_3 \qquad \Gamma \vdash E_3 <: E_2}{\Gamma \vdash t_1 \; ? \; t_2 : t_3 : E_2}$$

这直觉上是合理的，但它们对用于 FJ 的操作化语义的小步方式影响很小——类型保持性质实际上失败（很容易举一个例子来说明这一点）！

19.4.7 解答：奇怪，处理 super 比处理 self 难，因为我们需要某个方法来记住“当前执行的方法体”来自哪个类。这里至少有两种方式来完成它：

1. 用一些 super 引用应被查找的指示来注释项。
2. 增加整个类表被重写的预处理步骤，利用“拆分的”名字来说明它们来自哪个类，将 super 引用改为 this 引用。

19.5.1 解答：在给出主要证明之前，我们讨论一些必要的引理。和通常一样，关键的一个引理（A.14）与类型化和代换有关。

A.13 引理：如果 $mtype(m, D) = \overline{C} \to C_0$，那么对于所有 $C <: D$ 有 $mtype(m, C) = \overline{C} \to C_0$。

证明：通过直接归纳于推导 $C <: D$。注意，无论 m 是否在 $CT(C)$ 中定义，$mtype(m, C)$ 应和 $mtype(m, E)$ 相同，其中 $CT(C) = \text{class C extends E}\{\cdots\}$。

A.14 引理[项代换保持类型化]：如果 $\Gamma, \overline{x}:\overline{B} \vdash t:D$ 和 $\Gamma \vdash \overline{s}:\overline{A}$，其中 $\overline{A} <: \overline{B}$，那么对于一些 $C <: D$ 有 $\Gamma \vdash [\overline{x} \mapsto \overline{s}]t : C$。

证明：根据归纳于 $\Gamma, \overline{x}:\overline{B} \vdash t:D$ 推导。直觉与带子类型化的 lambda 演算十分相同；细节当然有一些变化。最有趣的情况是最后两个：

情况 T-VAR: $\quad t = x \qquad x:D \in \Gamma$

如果 $x \notin \overline{x}$，那么结果是显然的，因为 $[\overline{x} \mapsto \overline{s}]x = x$。另一方面，如果 $x = x_i$ 和 $D = B_i$，那么，由于 $[\overline{x} \mapsto \overline{s}]x = s_i$，令 $C = A_i$ 这种情况证毕。

情况 T-FIELD: $\quad t = t_0.f_i \qquad \Gamma, \overline{x}:\overline{B} \vdash t_0 : D_0 \qquad fields(D_0) = \overline{C}\;\overline{f} \qquad D = C_i$

通过归纳假设，有某满足 $\Gamma \vdash [\overline{x} \mapsto \overline{s}]t_0 : C_0$ 和 $C_0 <: D_0$ 的 C_0。对某 $\overline{D}\,\overline{g}$ 容易检查 $fields(C_0) = (fields(D_0), \overline{D}\,\overline{g})$。因此，由 T-Field，有 $\Gamma \vdash ([\overline{x} \mapsto \overline{s}]t_0).f_i : C_i$。

情况 T-INVK: $\quad t = t_0.m(\overline{t}) \qquad \Gamma, \overline{x}:\overline{B} \vdash t_0 : D_0 \qquad mtype(m, D_0) = \overline{E} \to D$
$\qquad \Gamma, \overline{x}:\overline{B} \vdash \overline{t} : \overline{D} \qquad \overline{D} <: \overline{E}$

通过归纳假设，存在 C_0 和 $\overline{C}$ 满足：

$$\Gamma \vdash [\overline{x} \mapsto \overline{s}]t_0 : C_0 \qquad C_0 <: D_0 \qquad \Gamma \vdash [\overline{x} \mapsto \overline{s}]\overline{t} : \overline{C} \qquad \overline{C} <: \overline{D}.$$

通过引理(A.13),$mytype(m, C_0) = \overline{E} \to D$。而且根据 <: 的传递性有 $\overline{C} <: \overline{E}$。因此,根据 T-Invk,$\Gamma \vdash [\bar{x} \mapsto \bar{s}]t_0 .m([\bar{x} \mapsto \bar{s}]\bar{t}) : D$。

情况 T-NEW: $t = \text{new } D(\bar{t}) \quad fields(D) = \overline{D}\ \bar{f} \quad \Gamma, \bar{x} : \overline{B} \vdash \bar{t} : \overline{C} \quad \overline{C} <: \overline{D}$

通过归纳假设,对 $\overline{E}$ 满足 $\overline{E} <: \overline{C}$ 的有 $\Gamma \vdash [\bar{x} \mapsto \bar{s}]\bar{t} : \overline{E}$。根据 <: 的传递性有 $\overline{E} <: \overline{D}$。因此,通过 T-Nen,$\Gamma \vdash (D[\bar{x} \mapsto \bar{s}]\bar{t}) : D$。

情况 T-UCAST: $t = (D)t_0 \quad \Gamma, \bar{x} : \overline{B} \vdash t_0 : C \quad C <: D$

通过归纳假设,有 E 满足 $\Gamma \vdash [\bar{x} \mapsto \bar{s}]t_0 : E$ 和 $E <: C$。我们根据 <: 的传递性有 $E <: D$,这由 T-Ucast 生成 $\Gamma \vdash (D)([\bar{x} \mapsto \bar{s}]t_0) : D$。

情况 T-DCAST: $t = (D)t_0 \quad \Gamma, \bar{x} : \overline{B} \vdash t_0 : C \quad D <: C \quad D \neq C$

通过归纳假设,有某 E 满足 $\Gamma \vdash [\bar{x} \mapsto \bar{s}]t_0 : E$ 和 $E <: C$。如果 $E <: D$ 或者 $D <: E$,那么分别根据 T-Ucast 或 T-Dcast 有 $\Gamma \vdash (D)([\bar{x} \mapsto \bar{s}]t_0) : D$。另一方面,如果有 $D \not<: E$ 和 $E \not<: D$,那么根据 T-Scast 有 $\Gamma \vdash (D)([\bar{x} \mapsto \bar{s}]t_0) : D$(带一个愚蠢警告)。

情况 T-SCAST: $t = (D)t_0 \quad \Gamma, \bar{x} : \overline{B} \vdash t_0 : C \quad D \not<: C \quad C \not<: D$

通过归纳假设,存在 E 满足 $\Gamma \vdash [\bar{x} \mapsto \bar{s}]t_0 : E$ 和 $E <: C$。这表示 $E \not<: D$(要理解这一点,注意每一个 FJ 中的类仅有一个超类。由此产生,如果有 $E <: C$ 和 $E <: D$,那么或者 $C <: D$ 或者 $D <: C$)。所以根据 T-Scast,有 $\Gamma \vdash (D)([\bar{x} \mapsto \bar{s}]t_0) : D$(带一个愚蠢警告)。

A.15 引理[弱化]: 如果 $\Gamma \vdash t : C$,那么 $\Gamma, x : D \vdash t : C$。

证明: 直接归纳。

A.16 引理: 如果 $mytype(m, C_0) = \overline{D} \to D$ 和 $mybody(m, C_0) = (\bar{x}, t)$,那么对某 D_0 和某 $C <: D$,有 $C_0 <: D_0$ 和 $\bar{x} : \overline{D}, this : D_0 \vdash t : C$。

证明: 根据归纳于 $mybody(m, C_0)$ 推导。基本情况(m 被定义在 C_0 之处)是简单的,因为 m 被定义在 $CT(C_0)$ 及一个良形式的类表暗示我们一定根据 T-Method 推导出 $\bar{x} : \overline{D}, this : C_0 \vdash t : C$。这个归纳步骤也是直接的。

我们现在准备好给出类型安全定理的证明。

定理(19.5.1)的证明: 根据 $t \to t'$ 的推导归纳,还有最终规则的情况分析。注意在 T-DCast 从结尾处算起的第二个子情况中会产生多么愚蠢的警告。

情况 E-PROJNEW: $t = \text{new } C_0(\bar{v}).f_i \quad t' = v_i \quad fields(C_0) = \overline{D}\ \bar{f}$

从 t 的形状可看出 $\Gamma \vdash t : C$ 推导的最终规则必为 T-Field,其中对某 D_0 有前提 $\Gamma \vdash \text{new } C_0(\bar{v}) : D_0$,且可看出 $C = D_i$。类似地,推导 $\Gamma \vdash \text{new } C_0(\bar{v}) : D_0$ 的最后规则必是 T-New,其中前提为 $\Gamma \vdash \bar{v} : \overline{C}$ 和 $\overline{C} <: \overline{D}$,且 $D_0 = C_0$,特别地,$\Gamma \vdash v_i : C_i$,由于 $C_i <: D_i$ 证毕。

情况 E-INVKNEW: $t = (\text{new } C_0(\bar{v})).m(\bar{u}) \quad t' = [\bar{u}/\bar{x}, \text{new } C_0(\bar{v})/\text{this}]t_0$
$mbody(m, C_0) = (\bar{x}, t_0)$

推导 $\Gamma \vdash t : C$ 的最终规则必定是 T-Invk 和 T-New,前提是:

$$\Gamma \vdash \text{new } C_0(\bar{v}) : C_0 \quad \Gamma \vdash \bar{u} : \overline{C} \quad \overline{C} <: \overline{D} \quad mtype(m, C_0) = \overline{D} \to C.$$

根据引理(A.16),对某 D_0 和 B 我们有:$\bar{x}:\bar{D}, this:D_0 \vdash t_0:B$,其中 $C_0 <: D_0$ 和 $B <: C$。根据引理(A.15),$\Gamma, \bar{x}:\bar{D}, this:D_0 \vdash t_0:B$,接着根据引理(A.14),对某 $E <: B$,有 $\Gamma \vdash [\bar{x} \mapsto \bar{u}, this \mapsto new\ C_0(\bar{v})]\ t_0:E$。通过<:的传递性,得到 $E <: C$,令 $C' = E$,这一情况证毕。

情况 E-CastNew: $t = (D)(new\ C_0(\bar{v}))$ $C_0 <: D$ $t' = new\ C_0(\bar{v})$

$\Gamma \vdash (D)(new\ C_0(\bar{v})):C$ 的证明必须结束于 T-UCast,因为结束于 T-SCast 或 T-DCast 将与假设 $C_0 <: D$ 矛盾。前提 T-UCast,给我们得出 $\Gamma \vdash new\ C_0(\bar{v}):C_0$ 和 $D = C$,这一情况证毕。同一规则的情况很简单。我们仅说明一个:

情况 RC-Cast: $t = (D)t_0$ $t' = (D)t_0'$ $t_0 \longrightarrow t_0'$

根据最后类型规则应用有三种子情况。

子情况 T-UCast: $\Gamma \vdash t_0 : C_0$ $C_0 <: D$ $D = C$

根据归纳假设,对某 $C_0' <: C_0$ 有 $\Gamma \vdash t_0':C_0'$。根据<:的传递性,有 $C_0' <: C$。因此,根据 T-UCast 有 $\Gamma \vdash (C)t_0':C$(没有附加的愚蠢警告)。

子情况 T-DCast: $\Gamma \vdash t_0 : C_0$ $D <: C_0$ $D = C$

根据归纳假设,对某 $C_0' <: C_0$ 有 $\Gamma \vdash t_0':C_0'$。如果 $C_0' <: C$ 或 $C <: C_0'$,那么根据 T-UCast 或 T-DCast,(不带任何附加愚蠢警告)有 $\Gamma \vdash (C)t_0':C$。另一方面,如果 $C_0' \not<: C$ 和 $C \not<: C_0'$,那么根据 T-SCast 有带愚蠢警告的 $\Gamma \vdash (C)t_0':C$。

子情况 T-SCast: $\Gamma \vdash t_0 : C_0$ $D \not<: C_0$ $C_0 \not<: D$ $D = C$

根据归纳假设,对某 $C_0' <: C_0$ 有 $\Gamma \vdash t_0':C_0'$。那么,同时有 $C_0' \not<: C$ 和 $C \not<: C_0'$。因此,有含愚蠢警告的结论 $\Gamma \vdash (C)t_0':C$。

20.1.1 解答:

```
Tree = μX. <leaf:Unit, node:{Nat,X,X}>;
leaf = <leaf=unit> as Tree;
▸ leaf : Tree

node = λn:Nat. λt₁:Tree. λt₂:Tree. <node={n,t₁,t₂}> as Tree;
▸ node : Nat → Tree → Tree → Tree

isleaf = λl:Tree. case l of <leaf=u> ⇒ true | <node=p> ⇒ false;
▸ isleaf : Tree → Bool

label = λl:Tree. case l of <leaf=u> ⇒ 0 | <node=p> ⇒ p.1;
▸ label : Tree → Nat

left = λl:Tree. case l of <leaf=u> ⇒ leaf | <node=p> ⇒ p.2;
▸ left : Tree → Tree

right = λl:Tree. case l of <leaf=u> ⇒ leaf | <node=p> ⇒ p.3;
▸ right : Tree → Tree
```

```
append = fix (λf:NatList→NatList→NatList.
               λl1:NatList. λl2:NatList.
               if isnil l1 then l2 else
               cons (hd l1) (f (tl l1) l2));
▸ append : NatList → NatList → NatList

preorder = fix (λf:Tree→NatList. λt:Tree.
                 if isleaf t then nil else
                 cons (label t)
                      (append (f (left t)) (f (right t))));
▸ preorder : Tree → NatList

t1 = node 1 leaf leaf;
t2 = node 2 leaf leaf;
t3 = node 3 t1 t2;
t4 = node 4 t3 t3;
l = preorder t4;
hd l;
▸ 4 : Nat

hd (tl l);
▸ 3 : Nat

hd (tl (tl l));
▸ 1 : Nat
```

20.1.2　解答：

```
fib = fix (λf: Nat→Nat→Stream. λm:Nat. λn:Nat. λ_:Unit.
          {n, f n (plus m n)}) 0 1;
▸ fib : Stream
```

20.1.3　解答：

```
Counter = μC. {get:Nat, inc:Unit→C, dec:Unit→C,
               reset:Unit→C, backup:Unit→C};
c = let create =
        fix (λcr: {x:Nat,b:Nat}→Counter. λs: {x:Nat,b:Nat}.
               {get    = s.x,
                inc    = λ_:Unit. cr {x=succ(s.x),b=s.b},
                dec    = λ_:Unit. cr {x=pred(s.x),b=s.b},
                backup = λ_:Unit. cr {x=s.x,b=s.x},
                reset  = λ_:Unit. cr {x=s.b,b=s.b}     })
    in create {x=0,b=0};
▸ c : Counter
```

20.1.4　解答：

```
D = μX. <nat:Nat, bool:Bool, fn:X→X>;

lam = λf:D→D. <fn=f> as D;
ap = λf:D. λa:D.
       case f of
         <nat=n> ⇒ divergeD unit
       | <bool=b> ⇒ divergeD unit
       | <fn=f> ⇒ f a;
ifd = λb:D. λt:D. λe:D.
        case b of
          <nat=n> ⇒ divergeD unit
        | <bool=b> ⇒ (if b then t else e)
        | <fn=f> ⇒ divergeD unit;
tru = <bool=true> as D;
fls = <bool=false> as D;
ifd fls one zro;
▸ <nat=0> as D : D

ifd fls one fls;
▸ <bool=false> as D : D
```

关注这个系统中能编码不良类型项的读者应该注意到我们已经做的是构造一个数据结构,用来在有递归类型的简单类型 lambda 演算的元语言中表示无类型项的对象语言。这种做法与(我们已经应用于全书章节实现中)将多种类型项及无类型项 lambda 演算表示为 ML 中的数据结构的做法一样令人吃惊。

```
lam = λf:D→D. <fn=f> as D;
ap = λf:D. λa:D. case f of
                   <nat=n> ⇒ divergeD unit
                 | <fn=f> ⇒ f a
                 | <rcd=r> ⇒ divergeD unit;

rcd = λfields:Nat→D. <rcd=fields> as D;
prj = λf:D. λn:Nat. case f of
                   <nat=n> ⇒ divergeD unit
                 | <fn=f> ⇒ divergeD unit
                 | <rcd=r> ⇒ r n;
myrcd = rcd (λn:Nat. if iszero 0 then zro
                     else if iszero (pred n) then one
                     else divergeD unit);
```

20.2.1 解答:这是一些在同构递归形式中更有趣的例子:

```
Hungry = μA. Nat → A;
f = fix (λf: Nat→Hungry. λn:Nat. fold [Hungry] f);
ff = fold [Hungry] f;
ff1 = (unfold [Hungry] ff) 0;
ff2 = (unfold [Hungry] ff1) 2;

fixT =
  λf:T→T.
    (λx:(μA.A→T). f ((unfold [μA.A→T] x) x))
    (fold [μA.A→T] (λx:(μA.A→T). f ((unfold [μA.A→T] x) x)))
```

```
D = μX. X→X;
lam = λf:D→D. fold [D] f;
ap = λf:D. λa:D. (unfold [D] f) a;

Counter = μC. {get:Nat, inc:Unit→C};
c = let create = fix (λcr: {x:Nat}→Counter. λs: {x:Nat}.
                       fold [Counter]
                         {get = s.x,
                          inc = λ_:Unit. cr {x=succ(s.x)}})
    in create {x=0};
c1 = (unfold [Counter] c).inc unit;
(unfold [Counter] c1).get;
```

21.1.7 解答:

$$\begin{array}{llllll} E_2(\emptyset) & = & \{a\} & E_2(\{a,b\}) & = & \{a,c\} \\ E_2(\{a\}) & = & \{a\} & E_2(\{a,c\}) & = & \{a,b\} \\ E_2(\{b\}) & = & \{a\} & E_2(\{b,c\}) & = & \{a,b\} \\ E_2(\{c\}) & = & \{a,b\} & E_2(\{a,b,c\}) & = & \{a,b,c\} \end{array}$$

E_2 闭包是$\{a\}$和$\{a,b,c\}$。E_2 一致是$\emptyset$,$\{a\}$和$\{a,b,c\}$。E_2 最小不动点是$\{a\}$。最大不动点是$\{a,b,c\}$。

21.1.9 解答:要证明对自然数一般归纳原则,我们如下进行。根据如下式子定义函数$F \in \mathcal{P}(\mathbb{N}) \to \mathcal{P}(\mathbb{N})$:

$$F(X) \quad = \quad \{0\} \cup \{i+1 \mid i \in X\}.$$

现在,假设我们有一个谓词(即一个数集)P 使 $P(0)$和 $P(i)$来表示 $P(i+1)$。那么,从 F 的定义,很容易看出 $F(P) \subseteq P$,即 P 是 F 闭包。根据归纳原理有 $\mu F \subseteq P$,但 μF 是自然数的整个集合(确实,这能被作为自然数集合的定义)。所以,对所有 $n \in \mathbb{N}$有$\mathcal{P}(n)$。

对字典序归纳,定义 $F \in \mathcal{P}(\mathbb{N} \times \mathbb{N}) \to \mathcal{P}(\mathbb{N} \times \mathbb{N})$为:

$$F(X) \quad = \quad \{(m,n) \mid \forall (m',n') < (m,n),\ (m',n') \in X\}$$

现在,假定我们有一个谓词(即一个自然数序对的集合)P,其中 $P(m',n')$对所有$(m',n') < (m,n)$仍然有 $P(m,n)$。像以前一样,从 F 的定义中,很容易看出 $F(P) \subseteq P$,即 P 是 F 闭包。根据归纳法原理,$\mu F \subseteq P$。为了结束说明,必须检查 μF 确实是自然数序对的集合(这是这一讨论中惟一的微妙点)。这能被放于两步讨论。首先,标记$\mathbb{N} \times \mathbb{N}$是 F 闭包(这可从 F 的定义直接得出);其次,说明没有合适的$\mathbb{N} \times \mathbb{N}$的子集是 F 闭包,或者说,$\mathbb{N} \times \mathbb{N}$是最小的 F 闭包。要理解这一点,可假定有一个较小的 F 闭包 Y,并令(m,n)是最小的不属于 Y 的序对;然后根据 F 的定义,得到 $F(Y) \not\subseteq Y$,即 Y 不是闭包,矛盾。

21.2.2 解答:定义一个树为一个部分函数 $T \in \{1,2\}^* \to \{\to, \times, \mathrm{Top}\}$ 且满足于以下约束:

- $T(\cdot)$有定义
- 如果 $T(\pi,\sigma)$有定义的,那么 $T(\pi)$也有定义

注意,树节点上符号→,×,Top 的存在完全是自由的,例如一个有 Top 的节点能有非凡的

子节点，等等。就像在21.2节中，我们重载了符号→，×，Top仍作为树上的操作符。

所有树的集合是全集$\mathcal{U}$，产生函数F是基于类型的熟悉的语法：

$$\begin{aligned} F(X) &= \{\text{Top}\} \\ &\cup \{T_1 \times T_2 \mid T_1, T_2 \in X\} \\ &\cup \{T_1 \to T_2 \mid T_1, T_2 \in X\}. \end{aligned}$$

从$\mathcal{T}$和$\mathcal{U}$的定义中可以看出$\mathcal{T} \subseteq \mathcal{U}$，所以在利益均等中比较集合是合理的，$\mathcal{T} = \nu F$和$\mathcal{T}_f = \mu F$。它接下来要检查等式为真的。

根据共归纳原则，$\mathcal{T} \subseteq \nu F$可从$\mathcal{T}$是$F$一致中推导出来。要得到$\nu F \subseteq \mathcal{T}$，需要检查对任何$T \in \nu F$，定义(21.2.1)中的最后两个条件。这能采用归纳于π的长度来完成。

根据$\mathcal{T}_f$是F闭包这一事实中的归纳法原则可得出$\mu F \subseteq \mathcal{T}_f$。要得到$\mathcal{T}_f \subseteq \mu F$，归纳于$T$的长度，$T \in \mathcal{T}_f$隐含$T \in \mu F$（$T \in \mathcal{T}_f$的长度能被定义为最长序列$\pi \in \{1,2\}^*$的长度，使$T(\pi)$有定义）。

21.3.3 解答：(Top，Top × Top)序对不在νS中。为理解这一点，根据S的定义可知这个序对对于任何X都不在$S(X)$中。所以这里没有任何S一致集合包含这个序对，特别是νS(为S一致)不包含它。

21.3.4 解答：用与νS而不是μS关联的树类型序对为例，我们能对任何无限类型T采用(T，T)序对。考虑序对集合$R = \{(T(\pi), T(\pi)) \mid \pi \in \{1,2\}^*\}$。对$S$定义检查很容易得出$R \subseteq S(R)$，而应用共归纳原则得出$R \subseteq \nu S$。那么由于$(T,T) \in R$有$(T,T) \in \nu S$。另一方面，因为$\mu S$只与有限类型关联，有$(T,T) \notin \mu S$这可以通过将$R'$作为所有有限类型序对的集合来建立，并根据归纳原则得到$\mu S \subseteq R'$。

只与νS_f相关联而与μS_f无关的有限类型序对(S，T)是不存在的，因为这两个不动点是一致的。这可由以下事实得出，对于任何$S, T \in \mathcal{T}_f$，$(S,T) \in \nu S_f$有$(S,T) \in \mu S_f$（由于T是有限树，后一个语句可对T进行归纳得出。还需要考虑T为Top，$T_1 \times T_2$，$T_1 \to T_2$的情况，检查S_f的定义，且使用等式$S_f(\nu S_f) = \nu S_f$和$S_f(\mu S_f) = \mu S_f$）。

21.3.8 解答：先定义树类型的恒等关系：$I = \{(T,T) \mid T \in \mathcal{T}\}$。如果我们能够显示$I$是$S$一致，接着共归纳原则将会告诉我们$I \subseteq \nu S$，也就是说，$\nu S$是自反的。要显示$I$的$S$一致，考虑一个元素$(T,T) \in I$，然后考虑T形式的情况。首先，假定T = Top。那么(T，T) = (Top，Top)，而(Top，Top)，根据定义，在$S(I)$中。接下来，假定$T = T_1 \times T_2$。那么由于$(T_1, T_1), (T_2, T_2) \in I$，$S$的定义给出$(T_1 \times T_2, T_1 \times T_2) \in S(I)$。对于$T = T_1 \to T_2$情况是类似的。

21.4.2 解答：根据共归纳原则，有$\mathcal{U} \times \mathcal{U}$是$F^{TR}$一致，或者说，$\mathcal{U} \times \mathcal{U} \subseteq F^{TR}(\mathcal{U} \times \mathcal{U})$。假定$(x,y) \in \mathcal{U} \times \mathcal{U}$。选任意$z \in \mathcal{U}$。那么$(x,z), (z,y) \in \mathcal{U} \times \mathcal{U}$，因此，根据$F^{TR}$的定义，同样可得到$(x,y) \in F^{TR}(\mathcal{U} \times \mathcal{U})$。

21.5.2 解答：要检查可逆性，我们仅检查S_f和S的定义，以确定每个$G_{(S,T)}$集合至多包含一个元素。

在 S_f 和 S 的定义中，每个子句明确指定一个可支持元素的形式和它所支持集合的内容，所以写下 $support_{S_f}$ 和 $support_S$ 是容易的[与定义(21.8.4)中的 S_m 的支持函数做对比]。

21.5.4　解答：

$$\frac{i \quad a}{h} \quad \frac{b \quad c}{a} \quad \frac{b}{d} \quad \frac{d}{e} \quad \frac{e}{b} \quad \frac{f \quad g}{c} \quad \frac{g}{f} \quad \frac{}{g}$$

21.5.6　解答：否，一个 $x \in \nu F \setminus \mu F$ 不一定导致在支持图中的一个循环：它同样能导致一个无穷链。例如：考虑被 $F(X) = \{0\} \cup \{n \mid n+1 \in X\}$ 定义的 $F \in \mathcal{P}(\mathbb{N}) \to \mathcal{P}(\mathbb{N})$。那么 $\mu F = \{0\}$ 及 $\nu F = \mathbb{N}$。同样，对任何 $n \in \nu F \setminus \mu F$，即对任意 $n > 0$，$support(n) = \{n+1\}$，且产生一个无穷链。

21.5.13　解答：首先，考虑部分正确性。根据对算法运行时的递归结构归纳来对每一部分进行证明。

1. 从 lfp 的定义，很容易看出 $lfp(X)$ 返回 $true$ 有两种情况。如果因为 $X = \varnothing$ 而使得 $lfp(X) = true$，我们有 $X \subseteq \mu F$。另一方面，如果因为 $lfp(support(X)) = ture$ 而使得 $lfp(X) = true$，那么，根据归纳假设，$support(X) \subseteq \mu F$，由此，引理(21.5.8)生成 $X \subseteq \mu F$。
2. 如果因为 $support(X)\uparrow$ 而使得 $lfp(X) = false$，那么根据引理(21.5.8)有 $X \not\subseteq \mu F$。否则，如果因为 $lfp(support(X)) = false$ 而得 $lfp(X) = false$，同时，根据归纳假设法，$support(X) \not\subseteq \mu F$，由引理(21.5.8)可得 $X \not\subseteq \mu F$。

接下来，我们将要描述产生函数 F 的特点，lfp 保证了 F 可以对所有有限的输入终止。对于此，一些新的术语是很有帮助的。对于一个给定的有限状态产生函数 $F \in \mathcal{P}(\mathcal{U}) \to \mathcal{P}(\mathcal{U})$，部分函数 $height_F \in \mathcal{U} \to \mathbb{N}$（或者仅是 $height$）是最小的满足以下条件的部分函数[①]：

$$height(x) = \begin{cases} 0 & \text{if } support(x) = \varnothing \\ 0 & \text{if } support(x)\uparrow \\ 1 + max\{height(y) \mid y \in support(x)\} & \text{if } support(x) \neq \varnothing \end{cases}$$

（注意，如果 x 本身参与可达循环或者依赖于该循环中的一个元素，$height(x)$ 就是无定义的）如果 $height_F$ 是一个全函数，那么产生函数 F 就被认为是有限高度的。容易检查到，如果 $y \in support(x)$ 且 $height(x)$ 和 $height(y)$ 有定义，那么 $height(y) < height(x)$。

现在，如果 F 是有限状态和有限高度，那么 $lfp(X)$ 对任何有限的输入集合 $X \subseteq \mathcal{U}$ 都可终止。为了理解这一点，知道由于 F 是有限状态，对由原始调用 $lfp(X)$ 产生的每一个递归调用 $lfp(Y)$，集合 Y 是有限的。由于 F 是有限高度，$h(Y) = max\{height(y) \mid y \in Y\}$ 是良定义的。因为 $h(Y)$ 随着每一个递归调用而减少，且总是非负的，它将作为 lfp 的终止手段。

21.8.5　解答：S_d 的定义与 S_m 的一样，除非最后一个子句中不包含条件 $T \neq \mu X.T_1$ 和 $T \neq Top$。为了判断 S_d 不是可逆的，可知集合 $G_{(}\mu X.Top, \mu Y.Top)$ 包含了两个产生集合 $\{(Top, \mu Y.Top)\}$ 和 $\{(\mu X.Top, Top)\}$（比较函数 S_m 的该集合中的内容）。

① 这种定义 height 的方法也可以改为表示部分函数关系的单调函数的最小不定点形式。

因为 S_d 和 S_m 的所有子句(除了最后一句)都是一样的,且 S_m 的最后一句是 S_d 的最后一句的约束,则包含 $\nu S_m \subseteq \nu S_d$ 是显而易见的。另一个包含 $\nu S_d \subseteq \nu S_m$,能用共归纳原则及下面的引理来证明,这个引理可建立 νS_d 是 S_m 一致的。

A.17 引理:对任意两种 μ 类型 S,T,如果(S,T)$\in \nu S_d$,那么(S,T)$\in S_m(\nu S_d)$。

证明概略:根据施归纳于 $k = \mu\text{-}height$(S)。这个归纳法证实了这一非正式的思想:即(S,T)$\in \nu S_d$的任一推导都可转变为同样情况中的另一个推导,而恰巧为(S,T)$\in \nu S_m$ 的推导。在左 μ 折叠规则中的限制要求被转变的推导有这样的性质,即每一个 μ 折叠规则的应用的序列都从一个左 μ 折叠序列开始,然后接着是一个右 μ 折叠序列。

21.9.2 解答:

$$\frac{}{\mathsf{T} \sqsubseteq \mathsf{T}} \qquad \frac{\mathsf{S} \sqsubseteq \mathsf{T_1}}{\mathsf{S} \sqsubseteq \mathsf{T_1} \times \mathsf{T_2}} \qquad \frac{\mathsf{S} \sqsubseteq \mathsf{T_2}}{\mathsf{S} \sqsubseteq \mathsf{T_1} \times \mathsf{T_2}}$$

$$\frac{\mathsf{S} \sqsubseteq \mathsf{T_1}}{\mathsf{S} \sqsubseteq \mathsf{T_1} \to \mathsf{T_2}} \qquad \frac{\mathsf{S} \sqsubseteq \mathsf{T_2}}{\mathsf{S} \sqsubseteq \mathsf{T_1} \to \mathsf{T_2}} \qquad \frac{\mathsf{S} \sqsubseteq [\mathsf{X} \mapsto \mu\mathsf{X.T}]\mathsf{T}}{\mathsf{S} \sqsubseteq \mu\mathsf{X.T}}$$

(注意,作为有趣的一点,产生函数 *TD* 与我们这里通篇考虑的产生函数是不同的:它是不可逆的。例如,$B \sqsubseteq A \times B \to B \times C$ 被两个集合$\{B \sqsubseteq A \times B\}$和$\{B \sqsubseteq B \times C\}$所支持,任意一个都不是另外一个的子集合)。

21.9.7 解答:所有关于 *BU* 的规则都与在练习 21.9.2 中解答中给出的关于 *TD* 的规则是相同的,除了开始于一个 μ 绑定器的类型规则外:

$$\frac{\mathsf{S} \preceq \mathsf{T}}{[\mathsf{X} \mapsto \mu\mathsf{X.T}]\mathsf{S} \preceq \mu\mathsf{X.T}}$$

21.11.1 解答:这里有许多。一个平凡的例子是,对几乎任意一个 T,有 μX.T 和[X $\mapsto$ μX.T]。更有趣的一点是 μX.Nat × (Nat × X)和 μX.Nat × *X*。

22.3.9 解答:这里是主要的算法约束产生规则:

$$\frac{\Gamma(\mathsf{x}) = \mathsf{T}}{\Gamma \vdash_F \mathsf{x} : \mathsf{T} \;|_F\; \{\}} \qquad \text{(CT-Var)}$$

$$\frac{\Gamma, \mathsf{x}{:}\mathsf{T_1} \vdash_F \mathsf{t_2} : \mathsf{T_2} \;|_{F'}\; C \qquad \mathsf{x} \notin dom(\Gamma)}{\Gamma \vdash_F \lambda\mathsf{x}{:}\mathsf{T_1}.\mathsf{t_2} : \mathsf{T_1} \to \mathsf{T_2} \;|_{F'}\; C} \qquad \text{(CT-Abs)}$$

$$\frac{\Gamma \vdash_F \mathsf{t_1} : \mathsf{T_1} \;|_{F'}\; C_1 \qquad \Gamma \vdash_{F'} \mathsf{t_2} : \mathsf{T_2} \;|_{F''}\; C_2 \qquad F'' = \mathsf{X}, F'''}{\Gamma \vdash_F \mathsf{t_1}\;\mathsf{t_2} : \mathsf{X} \;|_{F'''}\; C_1 \cup C_2 \cup \{\mathsf{T_1} = \mathsf{T_2} \to \mathsf{X}\}} \qquad \text{(CT-App)}$$

剩下的规则是相同的。原始规则的等价性和算法的表现形式叙述如下:

1. (可靠性)如果 $\Gamma \vdash_F$ t:T$|_F C$ 且在 Γ 和 t 中提到的变量没有出现在 *F* 中,那么 $\Gamma \vdash$ t:T$|_{F \setminus F} C$。
2. (完备性)如果 $\Gamma \vdash$ t:T$|_{\mathcal{X}} C$,那么在$\mathcal{X}$中存在一个名称置换 *F* 使得 $\Gamma \vdash_F$t:T$|_{\varnothing}$ C。

这两部分都可用归纳导出直接证明。对于第(1)部分中的情况应用,以下的引理是有用的:

如果在 Γ 和 t 中提到的类型变量没有出现在 F 中，且 $\Gamma \vdash_F \mathrm{t}:\mathrm{T} \mid_F C$，那么在 Γ 和 C 中提到的类型变量不会在 $F \setminus F'$ 中。

对于第(2)部分中相关的情况，下面的引理可用：

如果 $\Gamma \vdash_F \mathrm{t}:\mathrm{T} \mid_{F'} C$，那么 $\Gamma \vdash_{F,G} \mathrm{t}:\mathrm{T} \mid_{F',G} C$，其中 G 是任何一个新变量名字的序列。

22.3.10 解答：通过将约束集合表达为一个类型序对的列表，约束产生算法可以为练习 22.3.9 的解答中的推论规则的直接副本：

```
let rec recon ctx nextuvar t = match t with
      TmVar(fi,i,_) →
        let tyT = getTypeFromContext fi ctx i in
        (tyT, nextuvar, [])
    | TmAbs(fi, x, tyT1, t2) →
        let ctx' = addbinding ctx x (VarBind(tyT1)) in
        let (tyT2,nextuvar2,constr2) = recon ctx' nextuvar t2 in
        (TyArr(tyT1, tyT2), nextuvar2, constr2)
    | TmApp(fi,t1,t2) →
        let (tyT1,nextuvar1,constr1) = recon ctx nextuvar t1 in
        let (tyT2,nextuvar2,constr2) = recon ctx nextuvar1 t2 in
        let NextUVar(tyX,nextuvar') = nextuvar2() in
        let newconstr = [(tyT1,TyArr(tyT2,TyId(tyX)))] in
        ((TyId(tyX)), nextuvar',
         List.concat [newconstr; constr1; constr2])
    | TmZero(fi) → (TyNat, nextuvar, [])
    | TmSucc(fi,t1) →
        let (tyT1,nextuvar1,constr1) = recon ctx nextuvar t1 in
        (TyNat, nextuvar1, (tyT1,TyNat)::constr1)
    | TmPred(fi,t1) →
        let (tyT1,nextuvar1,constr1) = recon ctx nextuvar t1 in
        (TyNat, nextuvar1, (tyT1,TyNat)::constr1)
    | TmIsZero(fi,t1) →
        let (tyT1,nextuvar1,constr1) = recon ctx nextuvar t1 in
        (TyBool, nextuvar1, (tyT1,TyNat)::constr1)
    | TmTrue(fi) → (TyBool, nextuvar, [])
    | TmFalse(fi) → (TyBool, nextuvar, [])
    | TmIf(fi,t1,t2,t3) →
        let (tyT1,nextuvar1,constr1) = recon ctx nextuvar t1 in
        let (tyT2,nextuvar2,constr2) = recon ctx nextuvar1 t2 in
        let (tyT3,nextuvar3,constr3) = recon ctx nextuvar2 t3 in
        let newconstr = [(tyT1,TyBool); (tyT2,tyT3)] in
        (tyT3, nextuvar3,
         List.concat [newconstr; constr1; constr2; constr3])
```

22.3.11 解答：关于 fix 表达式的一个约束产生规则可直接从图 11.12 中的类型化规则 T-Fix 推导出来：

$$\frac{\Gamma \vdash \mathrm{t}_1 : \mathrm{T}_1 \mid_{\mathcal{X}_1} C_1 \qquad \mathrm{X} \text{ 不等于 } \mathcal{X}_1, \Gamma, \text{或 } \mathrm{t}_1}{\Gamma \vdash \mathtt{fix}\ \mathrm{t}_1 : \mathrm{X} \mid_{\mathcal{X}_1 \cup \{\mathrm{X}\}} C_1 \cup \{\mathrm{T}_1 = \mathrm{X} \to \mathrm{X}\}} \qquad \text{(CT-Fix)}$$

这个规则重构了 t_1 的类型(称它为 T_1)，确定 T_1 对一些新的 X 有形式 X→X，且生成一个 X 作为 fix t_1 的类型。

关于 letrec 表达式的约束产生规则及一个导出式的 letrec 定义,可以按顺序从这个规则中推导出来。

22.4.3 解答:

{X = Nat, Y = X→X}	[X ↦ Nat, Y ↦ Nat→Nat]
{Nat→Nat = X→Y}	[X ↦ Nat, Y ↦ Nat]
{X→Y = Y→Z, Z = U→W}	[X ↦ U→W, Y ↦ U→W, Z ↦ U→W]
{Nat = Nat→Y}	不可合一
{Y = Nat→Y}	不可合一
{ }	[ ]

22.4.6 解答:这个练习所需要的主要数据结构是一个代换的形式。这有几种选择;最简单的一个是重用练习 22.3.10 中的 constr 数据类型:一个代换仅是一个约束集合,整个集合的左端都是合一变量。如果我们定义一个函数 substinty 来将类型代换为单一类型变量:

```
let substinty tyX tyT tyS =
  let rec f tyS = match tyS with
      TyArr(tyS1,tyS2) → TyArr(f tyS1, f tyS2)
    | TyNat → TyNat
    | TyBool → TyBool
    | TyId(s) → if s=tyX then tyT else TyId(s)
  in f tyS
```

那么对于一个类型的整个代换的应用可以定义如下:

```
let applysubst constr tyT =
  List.fold_left
    (fun tyS (TyId(tyX),tyC2) → substinty tyX tyC2 tyS)
    tyT (List.rev constr)
```

这个合一函数仍然需要能够在某约束集合中将一个代换应用到所有类型:

```
let substinconstr tyX tyT constr =
  List.map
    (fun (tyS1,tyS2) →
       (substinty tyX tyT tyS1, substinty tyX tyT tyS2))
    constr
```

重要的仍然是检测循环依赖的"发生检验":

```
let occursin tyX tyT =
  let rec o tyT = match tyT with
      TyArr(tyT1,tyT2) → o tyT1 || o tyT2
    | TyNat → false
    | TyBool → false
    | TyId(s) → (s=tyX)
  in o tyT
```

这个合一函数现在是图 22.2 中给出的伪代码的一个直接副本。与通常一样,当合一失败时,它将一个文件位置和字符串作为附加的参数出现在打印错误信息中。

```
let unify fi ctx msg constr =
  let rec u constr = match constr with
      [] → []
    | (tyS,TyId(tyX)) :: rest →
        if tyS = TyId(tyX) then u rest
        else if occursin tyX tyS then
          error fi (msg ^ ": circular constraints")
        else
          List.append (u (substinconstr tyX tyS rest))
                      [(TyId(tyX),tyS)]
    | (TyId(tyX),tyT) :: rest →
        if tyT = TyId(tyX) then u rest
        else if occursin tyX tyT then
          error fi (msg ^ ": circular constraints")
        else
          List.append (u (substinconstr tyX tyT rest))
                      [(TyId(tyX),tyT)]
    | (TyNat,TyNat) :: rest → u rest
    | (TyBool,TyBool) :: rest → u rest
    | (TyArr(tyS1,tyS2),TyArr(tyT1,tyT2)) :: rest →
        u ((tyS1,tyT1) :: (tyS2,tyT2) :: rest)
    | (tyS,tyT)::rest →
        error fi "Unsolvable constraints"
  in
    u constr
```

这个合一的教学版本在打印有用的错误信息时的工作并不很有效。实际上,“解释”类型错误是在为具有类型重构功能的语言设计编译器时最复杂的部分之一。参考 Wand(1986)。

22.5.6 解答:通过扩展类型重构算法来处理记录并不是直接的,尽管可以做到。主要的困难在于不清楚记录投影需要产生什么样的约束。一个简单的最初尝试为:

$$\frac{\Gamma \vdash \mathtt{t} : \mathtt{T} \mid_{\mathcal{X}} C}{\Gamma \vdash \mathtt{t.l}_i : \mathtt{X} \mid_{\mathcal{X}\cup\{\mathtt{X}\}} C \cup \{\mathtt{T} = \{\mathtt{l}_i\mathtt{:X}\}\}}$$

但这并不令人满意,因为实际上,这条规则说明了 l_i 字段只能被包含了 l_i 字段而没有其他字段的记录所映射。

Wand(1987)提出了一种理想的解决方案,后来又被 Wand(1988,1989b),Remy(1989,1990)和其他人深入研究了。我们提出一种新的变量类型叫做行变量,其范围并不涉及类型而是涉及到字段标签和相关联的类型的“行”上。利用行变量,关于字段投影的约束产生规则可被写成:

$$\frac{\Gamma \vdash \mathtt{t}_0 : \mathtt{T} \mid_{\mathcal{X}} C}{\Gamma \vdash \mathtt{t}_0\mathtt{.l}_i : \mathtt{X} \mid_{\mathcal{X}\cup\{\mathtt{X},\sigma,\rho\}} C \cup \{\mathtt{T} = \{\rho\}, \rho = \mathtt{l}_i\mathtt{:X} \oplus \sigma\}} \quad \text{(CE-Proj)}$$

当 σ 和 ρ 是行变量且操作符包含了两行(假设它们的字段是不可连接的)。对项 $t.l_i$,如果 t 具有带字段 ρ 的记录类型,其中 ρ 含字段 l_i:X 及其他字段 σ,则 $t.l_i$ 的类型为 X。

这个改进算法的生成约束比具有原始重构算法的合一变量类型之间的简单等式集要复杂得多,因为新的约束集合仍然包括可结合交换的操作符。寻找这种约束集合的解决方案需要一种简单的等式合一模式。

23.4.3 解答:这里使用辅助的 append 的标准解答:

```
append = λX. (fix (λapp:(List X) → (List X) → (List X).
                  λl1:List X. λl2:List X.
                    if isnil [X] l1 then l2
                    else cons [X] (head [X] l1)
                                  (app (tail [X] l1) l2)));
```

```
▸ append : ∀X. List X → List X → List X
```

```
reverse =
  λX.
   (fix (λrev:(List X) → (List X).
          λl: (List X).
           if isnil [X] l
           then nil [X]
           else append [X] (rev (tail [X] l))
                           (cons [X] (head [X] l) (nil [X]))));
```

```
▸ reverse : ∀X. List X → List X
```

23.4.5 解答:

```
and = λb:CBool. λc:CBool.
        λX. λt:X. λf:X. b [X] (c [X] t f) f;
```

23.4.6 解答:

```
iszro = λn:CNat. n [Bool] (λb:Bool. false) true;
```

23.4.8 解答:

```
pairNat = λn1:CNat. λn2:CNat.
            λX. λf:CNat→CNat→X. f n1 n2;
fstNat = λp:PairNat. p [CNat] (λn1:CNat. λn2:CNat. n1);
sndNat = λp:PairNat. p [CNat] (λn1:CNat. λn2:CNat. n2);
```

23.4.9 解答:

```
zz = pairNat c0 c0;
f = λp:PairNat. pairNat (sndNat p) (cplus c1 (sndNat p));
prd = λm:CNat. fstNat (m [PairNat] f zz);
```

23.4.10 解答:

```
vpred = λn:CNat. λX. λs:X→X.
              λz:X.
                (n [(X→X)→X]
                   (λp:(X→X)→X. λq:(X→X). q (p s))
                   (λx:X→X. z))
                  (λx:X. x);
```

```
▸ vpred : CNat → CNat
```

感谢 Michael Levin 让我理解了这个例子。

23.4.11 解答:

```
head = λX. λdefault:X. λl:List X.
          l [X] (λhd:X. λtl:X. hd) default;
```

23.4.12 解答:这个练习最困难的部分是插入函数。这个解答是将给出的列表 l 应用到一个可以构建两个新列表的函数中,两个列表一个与原来的相同,另一个包含了 e。对于 l(从右到左)中的每一个元素 hd,这个函数被传给 hd 和已为 hd 右边元素构建的列表序对。新的列表序对通过将 e 与 hd 对比来构造:如果 e 小一些或二者相等,则它属于第二个结果列表的开始部分;于是,我们通过在第一个列表的前端(还不包含 e 的列表)添加一个 e 来建造第二个结果列表。另一方面,如果 e 比 hd 要大,那么它属于第二个列表的中间某个位置,且我们通过简单地将 hd 添加到已经构建好第二个列表中来生成一个新的第二个列表:

```
insert =
  λX. λleq:X→X→Bool. λl:List X. λe:X.
    let res =
      l [Pair (List X) (List X)]
        (λhd:X. λacc: Pair (List X) (List X).
           let rest = fst [List X] [List X] acc in
           let newrest = cons [X] hd rest in
           let restwithe = snd [List X] [List X] acc in
           let newrestwithe =
              if leq e hd
                then cons [X] e (cons [X] hd rest)
                else cons [X] hd restwithe in
           pair [List X] [List X] newrest newrestwithe)
        (pair [List X] [List X] (nil [X]) (cons [X] e (nil [X])))
    in snd [List X] [List X] res;
▸ insert : ∀X. (X→X→Bool) → List X → X → List X
```

接下来,我们需要一个关于数字的比较函数,由于使用原始数字,我们需要用 fix 来写它(可以在这里用 CNat 替代 Nat 来避免使用 fix)。

```
leqnat =
  fix (λf:Nat→Nat→Bool. λm:Nat. λn:Nat.
         if iszero m then true
         else if iszero n then false
         else f (pred m) (pred n));

▸ leqnat : Nat → Nat → Bool
```

最后,我们通过按照顺序将列表中的每一个元素插入一个新的列表来构建一个排序函数:

```
sort = λX. λleq:X→X→Bool. λl:List X.
         l [List X]
           (λhd:X. λrest:List X. insert [X] leq rest hd)
           (nil [X]);
▸ sort : ∀X. (X→X→Bool) → List X → List X
```

为了检测 sort 是否正确,我们构建一个无序列表:

```
l = cons [Nat] 9
      (cons [Nat] 2 (cons [Nat] 6 (cons [Nat] 4 (nil [Nat]))));
```

对它排序:

```
l = sort [Nat] leqnat l;
```

并且读出内容:

```
nth =
  λX. λdefault:X.
    fix (λf:(List X)→Nat→X. λl:List X. λn:Nat.
           if iszero n
             then head [X] default l
             else f (tail [X] l) (pred n));

▸ nth : ∀X. X → List X → Nat → X

nth [Nat] 0 l 0;

▸ 2 : Nat

nth [Nat] 0 l 1;

▸ 4 : Nat

nth [Nat] 0 l 2;

▸ 6 : Nat

nth [Nat] 0 l 3;

▸ 9 : Nat

nth [Nat] 0 l 4;

▸ 0 : Nat
```

一个良定义排序算法可以在 F 系统中实现,该示范就是 Reynolds(1985)的不朽之作。他的算法与这里讨论的有一些不同。

23.5.1 解答:证明的结构与定理(9.3.9)中的几乎完全一样。对于类型应用规则 E-TapTabs,我们需要一个附加的代换引理,与引理(9.3.8)并列。

If $\Gamma, X, \Delta \vdash t : T$, then $\Gamma, [X \mapsto S]\Delta \vdash [X \mapsto S]t : [X \mapsto S]T$.

为了得到一个有足够能力的归纳假设,这里需要一个额外的上下文 Δ;如果它被忽略,T-Abs 就会出现错误。

23.5.2 解答:再次,这个证明的结构同关于 $\lambda_{\rightarrow}$ 进展的证明[定理(9.3.5)]非常相似。规范的形式引理(9.3.4)可用一个附加的情况扩展:

如果 v 是类型 $\forall X.T_{12}$ 的一个值,那么 $v = \lambda X.T_{12}$。

这被应用在主要证明的类型应用情况中。

23.6.3 解答:当需要更多地了解来观察如何将各部分凑在一起并得到一个矛盾时,所有的部分,除了最后一个,都可直接归纳和/或可计算的证明。Pawel Urzyczyn 建议的证明结构如下:

(1) 对 t 直接归纳,使用关于类型化的逆转引理。

(2) 根据归纳于外部类型抽象和应用,有:

如果 t 对于 $\overline{Y}$,$\overline{B}$ 有形式 $\lambda\overline{Y}.(r[\overline{B}])$(其中 r 不必被揭示),同时如果 *erase*(t) = m 且 $\Gamma\vdash t:T$,那么存在一些形式为 $s=\lambda\overline{X}.(u[\overline{A}])$ 的类型 s,且 *erase*(s) = m 和 $\Gamma\vdash s:T$,更进一步的是 u 被揭示。

在基础的情况中,并不存在外部类型抽象或应用,即 r 自身是被揭示的,这样证明结束。

在归纳法情况中,r 的外部构造子是一个类型抽象或者一个类型应用。如果它是一个类型应用,即 $r_1[R]$,我们将 R 加入到序列 $\overline{B}$ 中且应用归纳假设。如果它是一个类型抽象,即 $\lambda Z.r_1$,那么这里就有两种子情况来考虑:

(a) 如果应用 $\overline{B}$ 序列是空的,那么我们可以将 Z 加入到抽象的序列 $\overline{Y}$ 中并应用归纳假设。

(b) 如果 $\overline{B}$ 是非空的,可将 t 写为:

$$t = \lambda\overline{Y}.\ ((\lambda Z.\ r_1)\ [B_0]\ [\overline{B}'])$$

其中 $\overline{B}=B_0\overline{B}'$。但这一次包括了一个 R-Beta2 的约式,约简这些约式得项:

$$t' = \lambda\overline{Y}.\ ([B_0 \mapsto Z]r_1\ [\overline{B}'])$$

其中$[B_0 \mapsto Z]r_1$ 包含了比 r 少的外部类型抽象和应用。更深入的是,主题归约定理告诉我们 t′与 t 有相同的类型。最终的结果可以通过应用归纳假设得到。

(3) 直接可从逆转引理得出。

(4) 从(1),(3)和(2)[两次]直接计算。

(5) 直接可从(2)和逆转引理得出。

(6) 根据归纳于 T_1 长度。在基本情况中,当 T_1 是一个变量,这个变量必须来自 $\overline{X}_1\overline{X}_2$,否则我们将有:

$$[\overline{X}_1\overline{X}_2 \mapsto \overline{A}](\forall\overline{Y}.T_1) \quad = \quad \forall\overline{Y}.W \quad = \quad \forall\overline{Z}.(\forall\overline{Y}.W)\to([\overline{X}_1 \mapsto \overline{B}]T_2),$$

但并不是这种情况(左端没有箭头且右端至少有一个)。另一些情况可直接从归纳假设法中得出。

(7) 假设,对于一个矛盾,omega 是可类型化的。那么,根据(1)和(3),存在揭示项 o = s u,其中:

$$\begin{array}{ll} \textit{erase}(s) = \lambda x.\ x\ x & \textit{erase}(u) = \lambda y.\ y\ y \\ \Gamma\vdash s : U\to V & \Gamma\vdash u : U. \end{array}$$

根据(2),存在项 $s'=\lambda\overline{R}.(s_0[\overline{E}])$ 和 $u'=\lambda\overline{V}.(u_0[\overline{F}])$,其中 s_0 和 u_0 为揭示的,且:

$$\begin{array}{ll} \textit{erase}(s_0) = \lambda x.\ x\ x & \textit{erase}(u_0) = \lambda y.\ y\ y \\ \Gamma\vdash s' : U\to V & \Gamma\vdash u' : U. \end{array}$$

因为 s′包含一个箭头,$\overline{R}$ 必须为空。同样,由于 s_0 和 u_0 为揭示的,它们必须从抽象开始,所以 $\overline{E}$ 和 $\overline{F}$ 仍然是空的,且我们有:

$$\begin{array}{lll} o' & = & s'\ u' \\ & = & s_0\ (\lambda\overline{V}.\ u_0) \\ & = & (\lambda x{:}T_x.\ w)\ (\lambda\overline{V}.\ \lambda y{:}T_y.\ v), \end{array}$$

其中 *erase*(w) = x x，且 *erase*(v) = y y 。根据逆转引理，$U = T_x$，且：

$\Gamma, x{:}T_X \vdash w : W \qquad \Gamma, \overline{V}, y{:}T_Y \vdash v : P.$

将(4)应用到其中的第一个，那么：

(a) $T_x = \forall \overline{X}.X_i$

(b) $T_x = \forall \overline{X}_1\overline{X}_2.T_1 \rightarrow T_2$ 且对于某 $\overline{A}$ 和 $\overline{B}$, $[\overline{X}_1\overline{X}_2 \mapsto A]T_1 = [\overline{X}_1 \mapsto B](\forall \overline{Z}.T_1 \rightarrow T_2)$。

根据(5)，T_x 必须有第二种形式，于是，根据(6)，T_1 最左边的叶子为 $X_i \in \overline{X}_1\overline{X}_2$。

现在，应用(4)到类型化 $\Gamma, \overline{V}, y{:}T_y \vdash v{:}P$ 中，将有：

(a) $T_y = \forall \overline{Y}.Y_i$

(b) $T_y = \forall \overline{Y}_1\overline{Y}_2.S_1 \rightarrow S_2$，且对于一些 $\overline{C}$ 和 $\overline{C}$，$[\overline{Y}_1\overline{Y}_2 \mapsto C]S_1 = [\overline{Y}_1 \mapsto D](\forall \overline{Z}'.S_1 \rightarrow S_2)$。

在前一个情况中，直接得出 T_y 最左边的叶子是 $Y_i \in \overline{Y}$。在后一个情况中，可以用(6)来再次得到 T_y 最左边的叶子是 $Y_i \in \overline{Y}_1\overline{Y}_2$。

但是，从 o′的形状和逆转引理中，得到：

$$\begin{aligned} \forall \overline{V}.T_y \rightarrow V &= T_x \\ &= \forall \overline{X}_1\overline{X}_2.T_1 \rightarrow T_2, \end{aligned}$$

所以，特别地，$T_y = T_1$。换句话说，T_1 的最左边的叶子与 T_y 的叶子相同。总之，我们有 $T_x = \forall \overline{X}_1\overline{X}_2.(\forall \overline{Y}.S) \rightarrow T_2$，且同时有 *leftmost-leaf*(S) = $X_i \in \overline{X}_1\overline{X}_2$ 和 *leftmost-leaf*(S) $= Y_i \in \overline{Y}$。由于变量 $\overline{X}_1\overline{X}_2$ 和 $\overline{Y}$ 被绑定在不同的地方，我们可以推导一个矛盾，我们最初假设 omega 是可类型化的一定是错误的。

23.7.1 解答：

```
let r = λX. ref (λx:X. x) in
(r[Nat] := (λx:Nat. succ x);
 (!(r[Bool])) true);
```

24.1.1 解答：包 p6 提供了一个常量 a 和一个函数 f，但这些分量的类型所允许的惟一操作是多次应用 f 到 a，然后抛开结果。包 p7 允许我们用 f 来产生类型 X 的值，但不能利用这些值做任何事。在包 p8 中，两个分量都是可用的，但现在已经没有什么隐藏的了，我们也能够同时放弃存在的包。

24.2.1 解答：

```
stackADT =
   {*List Nat,
    {new = nil [Nat],
     push = λn:Nat. λs:List Nat. cons [Nat] n s,
     top = λs:List Nat. head [Nat] s,
     pop = λs:List Nat. tail [Nat] s,
     isempty = isnil [Nat]}}
 as {∃Stack, {new: Stack, push: Nat→Stack→Stack, top: Stack→Nat,
              pop: Stack→Stack, isempty: Stack→Bool}};
▸ stackADT : {∃Stack,
              {new:Stack,push:Nat→Stack→Stack,top:Stack→Nat,
               pop:Stack→Stack,isempty:Stack→Bool}}
```

```
let {Stack,stack} = stackADT in
stack.top (stack.push 5 (stack.push 3 stack.new));
▸ 5 : Nat
```

24.2.2　解答:

```
counterADT =
   {*Ref Nat,
    {new = λ_:Unit. ref 1,
     get = λr:Ref Nat. !r,
     inc = λr:Ref Nat. r := succ(!r)}}
   as {∃Counter,
       {new: Unit→Counter, get: Counter→Nat, inc: Counter→Unit}};
▸ counterADT : {∃Counter,
                {new:Unit→Counter,get:Counter→Nat,
                 inc:Counter→Unit}}
```

24.2.3　解答:

```
FlipFlop = {∃X, {state:X, methods: {read: X→Bool, toggle: X→X,
                                    reset: X→X}}};
f = {*Counter,
     {state = zeroCounter,
      methods = {read = λs:Counter. iseven (sendget s),
                 toggle = λs:Counter. sendinc s,
                 reset = λs:Counter. zeroCounter}}}
    as FlipFlop;
▸ f : FlipFlop
```

24.2.4　解答:

```
c = {*Ref Nat,
     {state = ref 5,
      methods = {get = λx:Ref Nat. !x,
                 inc = λx:Ref Nat. (x := succ(!x); x)}}}
   as Counter;
```

24.2.5　解答:这个类型将允许用 union 方法来实现集合对象,但它阻止我们使用它们。为了调用这样一个对象的 union 方法,需要传给它两个与类型 X 形式非常相似的类型值。但这些值不能出自两个不同的集合对象,因为为了得到这两个状态需要打开每一个对象,且这将绑定两个不同的类型变量;第二个集合的状态不能被传给第一个集合的 union 操作(这并不是类型检查器的倔强所致:我们很容易看出将一个集合的具体的形式传给另一个集合的 union 操作是不合理的,因为第二个集合的表示形式总的来说可能和第一个完全不同)。所以,NatSet 类型的这个版本只允许它自身之间进行集合的联合操作!

24.3.2　解答:最低限度,我们需要说明类型化和计算规则由翻译所保持,或者说,如果我们为能完成所有这些翻译的函数记为⟦—⟧,那么 $\Gamma \vdash t:T$ 意味着 $⟦\Gamma⟧ \vdash ⟦t⟧ : ⟦T⟧$ 且 $t \to^* t'$ 意味着 $⟦t⟧ \to^* ⟦t'⟧$。这些性质检查起来很简单。我们可能希望发现,反过来也是对的。或者说,一个不良定义类型项在有存在类型的语言中,总能够通过翻译被映射到一个不良定义类型项,且一个已受阻项对应一个已受阻项;遗憾的是,这些性质都失败了:例

如,翻译将不良定义类型(和受阻)项({ * Nat,0} as {∃X.X})[Bool]映射到一个良定义类型(非受阻)的项。

24.3.3 解答: 我不知道这个问题从哪里得出。它好像是可能的,但这种变换不会是局部语法修饰——它需要立即被应用到整个程序中。

25.2.1 解答: 一个类型 T 在截参数 c 上的 d 步移位,记为 $\uparrow_c^d(T)$,定义如下:

$$
\begin{array}{lcl}
\uparrow_c^d(\mathsf{k}) & = & \begin{cases} \mathsf{k} & \text{if } k < c \\ \mathsf{k}+d & \text{if } k \geq c \end{cases} \\
\uparrow_c^d(\mathsf{T_1 \to T_2}) & = & \uparrow_c^d(\mathsf{T_1}) \to \uparrow_c^d(\mathsf{T_2}) \\
\uparrow_c^d(\forall.\mathsf{T_1}) & = & \forall.\ \uparrow_{c+1}^d(\mathsf{t_1}) \\
\uparrow_c^d(\{\exists,\mathsf{T_1}\}) & = & \{\exists,\ \uparrow_{c+1}^d(\mathsf{T_1})\}
\end{array}
$$

一个类型 T 中所有变量在截参 c 上的 d 步移位,记为 $\uparrow_0^d(T)$,或者说, $\uparrow^d(T)$

25.4.1 解答: 它为类型变量 X 提供了空间。鉴于原始的 v_{12} 的定义只与 T 有关联,用 t_2 代换 v_{12} 的结果在 Γ,X 形式的上下文中被假设为良辖定的。

26.2.3 解答: 需要全 $F_{<:}$ 规则的一个地方在 Abadi,Cardelli 和 Viswanathan(1996)的对象编码之中。在 Abadi 和 Cardelli(1996)中也有描述。

26.3.4 解答:

```
spluszz = λn:SZero. λm:SZero.
              λX. λS<:X. λZ<:X. λs:X→S. λz:Z.
                n [X] [S] [Z] s (m [X] [S] [Z] s z);

spluspn = λn:SPos. λm:SNat.
              λX. λS<:X. λZ<:X. λs:X→S. λz:Z.
                n [X] [S] [X] s (m [X] [S] [Z] s z);

▸ spluspn : SPos → SNat → SPos
```

26.3.5 解答:

```
SBool  = ∀X. ∀T<:X. ∀F<:X. T→F→X;
STrue  = ∀X. ∀T<:X. ∀F<:X. T→F→T;
SFalse = ∀X. ∀T<:X. ∀F<:X. T→F→F;
tru = λX. λT<:X. λF<:X. λt:T. λf:F. t;

▸ tru : STrue

fls = λX. λT<:X. λF<:X. λt:T. λf:F. f;

▸ fls : SFalse

notft = λb:SFalse. λX. λT<:X. λF<:X. λt:T. λf:F. b[X][F][T] f t

▸ notft : SFalse → STrue

nottf = λb:STrue. λX. λT<:X. λF<:X. λt:T. λf:F. b[X][F][T] f t;

▸ nottf : STrue → SFalse
```

26.4.3　解答：在第(1)和第(2)部分中的抽象和类型抽象情况中及第(3)和第(4)部分中的量词情况中。

26.4.5　解答：对子类型化推导归纳讨论第(1)部分。所有的情况或者是立即得出(S-Refl，S-Top)或者是归纳假设(S-Trans，S-Arrow，S-All)的直接应用，除了 S-TVar，S-TVar 更为有趣。假设 $\Gamma, X <: Q, \Delta \vdash S <: T$ 的推导中的最后条规则是 S-TVar 的一个实例，或者说，S 是某变量 Y，T 是上下文中 Y 的上界。这里有两种可能性要考虑。如果 X 和 Y 是不同的变量，那么 $Y <: T$ 的假设仍能在上下文 $\Gamma, X <: P, \Delta$ 中找出，结果立即得出。在另一方面，如果 $X = Y$，那么 $T = Q$；为了完成这个证明，我们需要显示 $\Gamma, X <: P, \Delta \vdash X <: Q$。通过 S-TVar，我们有 $\Gamma, X <: P, \Delta \vdash X <: P$。更多地，根据假设 $\Gamma \vdash P <: Q$，那么，根据弱化引理(26.4.2)，$\Gamma, X <: P, \Delta \vdash P <: Q$。将这两个新的推导结合 S-TVar，就可以得出想要的结果。
第(2)部分是常规归纳类型化推导，使用第(1)部分作为类型应用情况的子类型化的前提。

26.4.11　解答：所有的证明都是根据直接归纳子类型化推导的。通过推导中最后规则的情况分析只说明第一个。S-Refl 和 S-Top 的情况是显而易见的。S-TVar 不可能发生(S-TVar 结论的左端只可能是一个变量，不可能是一个箭头)；同样，S-All 不可能发生。如果最后规则是 S-Arrow 的一个实例，那么子推导就是我们想要的结果。假设最后的规则是 S-Trans 的一个实例，或者说我们对某 U 有 $\Gamma \vdash S_1 \to S_2 <: U$ 和 $\Gamma \vdash U <: T$。根据归纳法假设，或者 U 是 Top[根据练习的第(4)部分和我们已经完成的工作，T 也是 Top]或者 U 有形式 $U_1 \to U_2$ 其中 $\Gamma \vdash U_1 <: S_1$ 和 $\Gamma \vdash S_2 <: U_2$。在后一种情况中，我们将归纳假设再次应用到原始的 S-Trans 的第二个子推导中来得到，或者 $T = Top$(我们已完成)，或者 T 有形式 $T_1 \to T_2$ 其中 $\Gamma \vdash T_1 <: U_1$ 和 $\Gamma \vdash U_2 <: T_2$。传递性的两个应用告诉我们 $\Gamma \vdash T_1 <: S_1$ 且 $\Gamma \vdash S_2 <: T_2$，据此想要的结果可从 S-Arrow 中推出。

26.5.1　解答：

$$\frac{\Gamma \vdash S_1 <: T_1 \qquad \Gamma, X{<:}S_1 \vdash S_2 <: T_2}{\Gamma \vdash \{\exists X{<:}S_1, S_2\} <: \{\exists X{<:}T_1, T_2\}} \quad \text{(S-Some)}$$

26.5.2　解答：没有子类型化，就仅有 4 个：

```
{*Nat, {a=5,b=7}} as {∃X, {a:Nat,b:Nat}};
{*Nat, {a=5,b=7}} as {∃X, {a:X,b:Nat}};
{*Nat, {a=5,b=7}} as {∃X, {a:Nat,b:X}};
{*Nat, {a=5,b=7}} as {∃X, {a:X,b:X}};
```

结合子类型化和囿量词，那就确实多了一些，例如：

```
{*Nat, {a=5,b=7}} as {∃X, {a:Nat}};
{*Nat, {a=5,b=7}} as {∃X, {b:X}};
{*Nat, {a=5,b=7}} as {∃X, {a:Top,b:X}};
{*Nat, {a=5,b=7}} as {∃X, Top};
{*Nat, {a=5,b=7}} as {∃X<:Nat, {a:X,b:X}};
{*Nat, {a=5,b=7}} as {∃X<:Nat, {a:Top,b:X}};
```

26.5.3 解答:完成本题的一个途径是将重置计数器 ADT 嵌套在计数器 ADT 之内:

```
counterADT =
   {*Nat,
    {new = 1, get = λi:Nat. i, inc = λi:Nat. succ(i),
     rcADT =
       {*Nat,
        {new = 1, get = λi:Nat. i, inc = λi:Nat. succ(i),
         reset = λi:Nat. 1}}
       as {∃ResetCounter<:Nat,
        {new: ResetCounter, get: ResetCounter→Nat,
         inc: ResetCounter→ResetCounter,
         reset: ResetCounter→ResetCounter}} }}
     as {∃Counter,
         {new: Counter, get: Counter→Nat, inc: Counter→Counter,
          rcADT:
            {∃ResetCounter<:Counter,
             {new: ResetCounter, get: ResetCounter→Nat,
              inc: ResetCounter→ResetCounter,
              reset: ResetCounter→ResetCounter}}}};
▸ counterADT : {∃Counter,
                  {new:Counter,get:Counter→Nat,inc:Counter→Counter,
                   rcADT:{∃ResetCounter<:Counter,
                           {new:ResetCounter,get:ResetCounter→Nat,
                            inc:ResetCounter→ResetCounter,
                            reset:ResetCounter→ResetCounter}}}}
```

当这些包都被打开,结果就是用于检查程序中剩余部分的上下文将包含形为 Counter <: Top, counter: {⋯}, ResetCounter <: Counter, resetCounter: {⋯} 的绑定的类型变量:

```
let {Counter,counter} = counterADT in
let {ResetCounter,resetCounter} = counter.rcADT in
counter.get
  (counter.inc
     (resetCounter.reset (resetCounter.inc resetCounter.new)));
▸ 2 : Nat
```

26.5.4 解答:所有需要我们做的就是在 24.3 节的编码中的显著位置上添加界限。在类型的层次上,我们得到:

$$\{\exists X{<:}S, T\} \stackrel{\text{def}}{=} \forall Y.\ (\forall X{<:}S.\ T \to Y) \to Y.$$

在项的层次上的改变可从这里直接看出。

27.1 解答:这里有一种方法:

```
setCounterClass =
  λM<:SetCounter. λR<:CounterRep.
  λself: Ref(R→M).
    λr: R.
       {get = λ_:Unit. !(r.x),
        set = λi:Nat.  r.x:=i,
        inc = λ_:Unit. (!self r).set (succ((!self r).get unit))};
```

```
▸ setCounterClass : ∀M<:SetCounter.
                      ∀R<:CounterRep.
                        (Ref (R→M)) → R → SetCounter

  instrCounterClass =
    λM<:InstrCounter.
    λR<:InstrCounterRep.
    λself: Ref(R→M).
      λr: R.
        let super = setCounterClass [M] [R] self in
           {get = (super r).get,
            set = λi:Nat. (r.a:=succ(!(r.a)); (super r).set i),
            inc = (super r).inc,
            accesses = λ_:Unit. !(r.a)};

▸ instrCounterClass : ∀M<:InstrCounter.
                        ∀R<:InstrCounterRep.
                          (Ref (R→M)) → R → InstrCounter

  newInstrCounter =
    let m = ref (λr:InstrCounterRep. error as InstrCounter) in
    let m' =
      instrCounterClass [InstrCounter] [InstrCounterRep] m in
    (m := m';
     λ_:Unit. let r = {x=ref 1, a=ref 0} in m' r);

▸ newInstrCounter : Unit → InstrCounter
```

28.2.3 解答:在 T-Tabs 情况中,我们添加一个 S-Refl 的平凡应用来作为 S-All 的额外的前提。在 T-Tapp 情况中,子类型逆转引理(对全 $F_{<:}$)告诉我们 $N_1 = \forall X <: N_{11}.N_{12}$,同时 $\Gamma \vdash T_{11} <: N_{11}$ 且 $\Gamma, X <: T_{11} \vdash N_{12} <: T_{12}$。使用传递性,可以得到 $\Gamma \vdash T_2 <: T_{11}$,其证明了使用 TA-Tapp 来得到 $\Gamma \mapsto t_1[T_2]:[X \mapsto T_2]N_{12}$。可像以前一样利用在代换之下[引理(26.4.8)]的子类型化的保持性质来得到 $\Gamma \vdash [X \mapsto T_2]N_{12} <: [X \mapsto T_2]T_{12} = T$,并以此来结束。

28.5.1 解答:定理(28.3.5)(特别是在 S-All 情况中)对全 $F_{<:}$ 是无效的。

28.5.6 解答:注意,首先,囿和非囿量词不应该被混合:必须有一个子类型化规则来比较两个囿量词,而另一个来比较非囿量词,但不需要比较囿量词和非囿量词的规则。否则我们就要回到原地重新开始!

对于第(1)部分和第(2)部分,详情请看 Katiyar 和 Sankar(1992)。对于第(3)部分,答案是否定的:将具有广度子类型化的记录类型加入到被限制的系统中将使它再次变为不确定的。问题出在空的记录类型是一种最大类型(在记录类型中)。而且在使用 Ghelli 例子的改进版本的子类型化检查器中,它将再次造成发散。如果 $T = \forall X <: \{\}.\neg\{a: \forall Y <: X.\neg Y\}$,那么输入 $X_0 <: \{a:T\} \vdash X_0 <: \{a: \forall X_1 <: X_0.\neg X_1\}$ 将导致子类型检查器出现发散。

Martin Hofmann 帮助我们解答这个例子。Katiyar 和 Sankar(1992)也做出了相同的工作。

28.6.3　解答：

1. 我计算出了 9 种公用子类型：

$$
\begin{array}{lll}
\forall X{<:}Y'{\to}Z.\ Y{\to}Z' & \forall X{<:}Y'{\to}Z.\ Top{\to}Z' & \forall X{<:}Y'{\to}Z.\ X \\
\forall X{<:}Y'{\to}Top.\ Y{\to}Z' & \forall X{<:}Y'{\to}Top.\ Top{\to}Z' & \forall X{<:}Y'{\to}Top.\ X \\
\forall X{<:}Top.\ Y{\to}Z' & \forall X{<:}Top.\ Top{\to}Z' & \forall X{<:}Top.\ X.
\end{array}
$$

2. 不仅 $\forall X <: Y'\to Z.Y\to Z'$，而且 $\forall X <: Y'\to Z.X$ 是 S 和 T 的下界，但这两种类型并没有同时是 S 和 T 子类型的公用子类型。
3. 考虑 $S\to Top$ 和 $T\to Top$（或者 $\forall X <: Y'\to Z.Y\to Z'$ 和 $\forall X <: Y'\to Z.X.$）。

28.7.1　解答：在图 A.2 中定义的函数 $R_{X,\Gamma}$ 和 $L_{X,\Gamma}$ 将类型分别映射到它们最小的 X-free 超类型和最大的 X-free 子类型中（为了避免散乱，省略了下标 X 和 Γ）。两个定义有不同的附属条件，因为，当 L 出现时，我们需要检查它是否已被定义（写为 $L(T)\neq fail$），同时 R 总是被定义的，幸亏有了 Top 类型。Ghelli 和 Pierce（1998）证明了这些定义的正确性。

$$
\begin{array}{lcl}
R(\forall Y{<:}S.T) & = & \begin{cases} \forall Y{<:}S.\ R(T) & \text{if } X\notin FVS \\ Top & \text{if } X\in FV(S) \end{cases} \\
R(\{\exists Y{<:}S.T\}) & = & \begin{cases} \{\exists Y{<:}S, R(T)\} & \text{if } X\notin FV(S) \\ Top & \text{if } X\in FV(S) \end{cases} \\
R(S\to T) & = & \begin{cases} L(S)\to R(T) & \text{if } L(S)\neq fail \\ Top & \text{if } L(S)= fail \end{cases} \\
R(X) & = & T \quad \text{where } X{<:}T\in\Gamma \\
R(Y) & = & Y \quad \text{when } Y\neq X \\
R(Top) & = & Top
\end{array}
$$

$$
\begin{array}{lcl}
L(\forall Y{<:}S.T) & = & \begin{cases} \forall Y{<:}S.\ L(T) & \text{if } L(T)\neq fail \text{ and } X\notin FV(S) \\ fail & \text{otherwise} \end{cases} \\
L(\{\exists Y{<:}S.T\}) & = & \begin{cases} \{\exists Y{<:}S.\ L(T)\} & \text{if } L(T)\neq fail \text{ and } X\notin FV(S) \\ fail & \text{otherwise} \end{cases} \\
L(S\to T) & = & \begin{cases} R(S)\to L(T) & \text{if } L(T)\neq fail \\ fail & \text{if } L(T)= fail \end{cases} \\
L(X) & = & fail \\
L(Y) & = & Y \quad \text{when } Y\neq X \\
L(Top) & = & Top
\end{array}
$$

图 A.2　一个给定类型的最小 X-free 超类型和最大 X-free 子类型

28.7.2　解答：显示全囿存在类型的不确定性的一种简单方法（根据 Ghelli 和 Pierce，1998）是从全 $F_{<:}$ 中的子类型化问题中给出一个翻译 $[\![-]\!]$ 到只有存在类型系统的子类型化问题中，使得当且仅当 $[\![\Gamma\vdash S<:T]\!]$ 在有存在类型的系统中是可证明的，$\Gamma\vdash S<:T$ 在 $F_{<:}$ 中也是可证明的。这种编码在类型上可以被定义为：

$$
\begin{array}{lcllcl}
[\![X]\!] & = & X & [\![Top]\!] & = & Top \\
[\![\forall X{<:}T_1.T_2]\!] & = & \neg\{\exists X{<:}T_1,\neg[\![T_2]\!]\} & [\![T_1\to T_2]\!] & = & [\![T_1]\!]\to[\![T_2]\!]
\end{array}
$$

其中 $\neg S=\forall X<:S.X$。通过 $[\![X_1<:T_1,\cdots,X_n<:T_n]\!]=X_1<:[\![T_1]\!],\cdots,X_n<:[\![T_n]\!]$ 类将它扩展到上下文当中，同时通过 $[\![\Gamma\vdash S<:T]\!]=[\![\Gamma]\!]\vdash[\![S]\!]<:[\![T]\!]$，将它扩展到子类型化语句中。

29.1.1　解答：$\forall X.X \to X$ 是一个合适类型，包含元素如 $\lambda X.\lambda x:X.x$。这些项是多态的函数，当用一个类型 T 实例化时，会生成一个从 T 到 T 的函数。相比之下，$\lambda X.X \to X$ 是一个类型操作子——一个函数，当应用到类型 T 时，生成适当的从 T 到 T 函数的类型 $T \to T$。
换句话说：$\forall X <: X \to X$ 是一个元素为从类型到项的项级函数的类型；实例化其中的一个（通过将其应用到一个类型中，写做 t[T]）来生成一个箭头类型 $T \to T$ 的元素。另一方面，$\lambda X.X \to X$ 自身是一个函数（从类型到类型）；通过一个类型 T 来实例化它（写做 $(\lambda X.X \to X)T$）生成类型 $T \to T$ 自身，而不是它的一个元素。
例如，如果 fn 有类型 $\forall X.X \to X$ 且 $Op = \lambda X.X \to X$，那么 $fn[T] : T \to T = Op\ T$。

29.1.2　解答：Nat →Nat 是一种（合适的）函数类型，而不是一个类型级的函数。

30.3　解答：引理（30.3.1）被应用在 T-Abs，T-Tapp，T-Eq 中。引理（30.3.2）被应用在 T-Var 中。

30.3.8　解答：对给定推导的总体长度进行归纳，并结合两者的最后规则进行情况分析。如果有一个推导以 QR-Refl 结束，那么另一个推导就是希望得出的结果。如果有一个推导以 QR-Abs，QR-Arrow 或 QR-All 结束，那么根据规则的形式，两个推导都必须以同样的规则结束，且结果可得出。如果两个推导都结束于QR-App，那么结果也可直接通过归纳假设得出。剩下的可能性更有趣。
如果两个推导都以 QR-AppAbs 结束，那么我们有：

$$S = (\lambda X{::}K_{11}.S_{12})\ S_2 \qquad T = [X \mapsto T_2]T_{12} \qquad U = [X \mapsto U_2]U_{12},$$

且：

$$S_{12} \Rrightarrow T_{12} \qquad S_2 \Rrightarrow T_2 \qquad S_{12} \Rrightarrow U_{12} \qquad S_2 \Rrightarrow U_2.$$

通过归纳假设，有 V_{12} 和 V_2，型如：

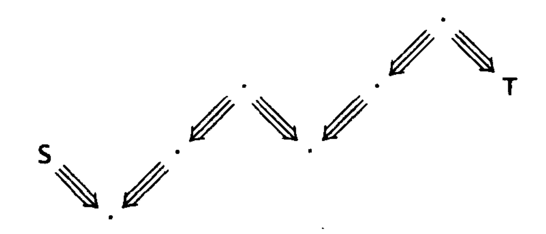
$$T_{12} \Rrightarrow V_{12} \qquad T_2 \Rrightarrow V_2 \qquad U_{12} \Rrightarrow V_{12} \qquad U_2 \Rrightarrow V_2.$$

通过两次利用引理（30.3.7），我们得到 $[X \mapsto T_2]T_{12} \Rrightarrow [X \mapsto V_2]V_{12}$ 和 $[X \mapsto U_2]U_{12} \Rrightarrow [X \mapsto V_2]V_{12}$，也就是，$T \Rrightarrow V$ 且 $U \Rrightarrow V$。
最后，假设一个推导（例如第一个）以 QR-App 来结束，而另一个以 QR-AppAbs 结束。在这种情况下，我们有：

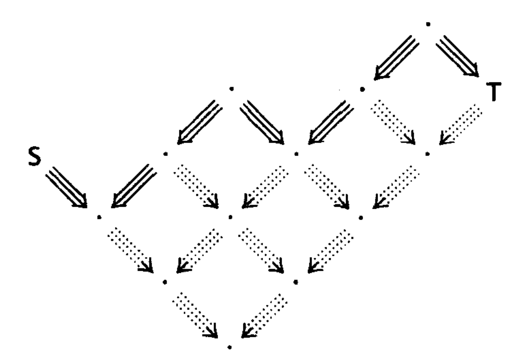
$$S = (\lambda X{::}K_{11}.S_{12})\ S_2 \qquad T = (\lambda X{::}K_{11}.T'_{12})\ T'_2 \qquad U = [X \mapsto U_2]U_{12},$$

其中 $S_{12} \Rrightarrow T_{12}$，$S_2 \Rrightarrow T_2$，$S_{12} \Rrightarrow U_{12}$ 且 $S_2 \Rrightarrow U_2$。还有，根据归纳假设法，有 V_{12} 和 V_2 型如 $T_{12} \Rrightarrow V_{12}$，$T_2 \Rrightarrow V_2$，$U_{12} \Rrightarrow V_{12}$ 且 $U_2 \Rrightarrow V_2$。将规则 QR-AppAbs 应用于第一个和第二个导出，同时将引理（30.3.7）应用于第三个和第四个，可以得到 $T \Rrightarrow V$ 或 $U \Rrightarrow V$。

30.3.10　解答：首先观察到我们可以重新组织 $S \Lleftarrow\!\!\Rrightarrow^* T$ 的任一个推导，这样对称性和传递性都不能用在对称性规则的一个实例的子推导中，也就是说，我们可以利用一个与 QR-Trans 结合的步骤序列来从 S 到 T 中得到，当每一个步骤都由一个单一步骤的归约组成，且可能其后跟随着一个单一的对称性实例。这个序列可显示如下：



(从右指向左的箭头是以对称性结束的归约,同时从左到右的箭头是非对称性的归约)。现在利用引理(30.3.8)反复地向本图中的底部添加小的菱形,直到我们达到通常的 S 和 T 的约简:



同样的证明可用一个标准的归纳形式来表示,不需要图片,但如果不用图片不能使它更令人信服,这样就很可能更难于理解。

30.3.17 解答: 如果添加第一条奇怪的规则,进展的性质将不满足;然而,保持性质能满足。如果添加了第二条规则,不但进展,而且保持都会不满足。

30.3.20 解答: 将你的解决方案和 fomega 检查器的源代码做对比。

30.5.1 解答: 我们用参数化类型簇 List T n 代替类型簇 FloatList n,且具有以下操作:

```
nil   :  ∀X. FloatList X 0
cons  :  ∀X. Πn:Nat. X → FloatList X (succ n)
hd    :  ∀X. Πn:Nat. List X (succ n) → X
tl    :  ∀X. Πn:Nat. List X (succ n) → List X n
```

31.2.1 解答:

Γ	⊢	A	<:	Id B	*Yes*
Γ	⊢	Id A	<:	B	*Yes*
Γ	⊢	λX.X	<:	λX.Top	*Yes*
Γ	⊢	λX. ∀Y<:X. Y	<:	λX. ∀Y<:Top. Y	*No*
Γ	⊢	λX. ∀Y<:X. Y	<:	λX. ∀Y<:X. X	*Yes*
Γ	⊢	F B	<:	B	*Yes*
Γ	⊢	B	<:	F B	*No*
Γ	⊢	F B	<:	F B	*Yes*
Γ	⊢	∀F<:(λY.Top→Y). F A	<:	∀F<:(λY.Top→Y). Top→B	*Yes*
Γ	⊢	∀F<:(λY.Top→Y). F A	<:	∀F<:(λY.Top→Y). F B	*No*
Γ	⊢	Top[*⇒*]	<:	Top[*⇒*⇒*]	*No*

32.5.1　解答:关键一点是 Object M 是一个存在类型:Object 是操作子:

```
λM::*⇒*. {∃X, {state:X, methods:M X}}
```

的缩写。当我们将此应用于 M 时,可得到一个约式,该约式可约简为存在类型:

```
{∃X, {state:X, methods:M X}}.
```

注意,这个转换中没有包含关系(因为没有信息丢失)。

32.5.2　解答:

```
sendget =
  λM<:CounterM. λo:Object M.
    let {X, b} = o in b.methods.get(b.state);

sendreset =
  λM<:ResetCounterM. λo:Object M.
    let {X, b} = o in
      {*X,
       {state = b.methods.reset(b.state),
        methods = b.methods}}  as Object M;
```

32.7.2　解答:最小的类型化属性对我们在定义它时采用的演算是错误的。考虑这个项:

```
{#x={a=5,b=7}}.
```

可给出类型{#x:{a:Nat}}和{#x:{a:Nat,b:Nat}},但这些类型是不可比较的。一个合理的解决办法是明确地在记录项中用其理想的类型注释不变的字段。这样可以有效地将在上面两个类型之间做出选择的权力交给程序员。

32.9.1　解答:

```
MyCounterM =
  λR. {get: R→Nat, set:R→Nat→R, inc:R→R, accesses:R→Nat,
           backup:R→R, reset:R→R};

MyCounterR = {#x:Nat,#count:Nat,#old:Nat};

myCounterClass =
   λR<:MyCounterR.
   λself: Unit→MyCounterM R.
   λ_:Unit.
      let super = instrCounterClass [R] self unit in
      {get = super.get,
       set = super.set,
       inc = super.inc,
       accesses = super.accesses,
       reset = λs:R. s←x=s.old,
       backup = λs:R. s←old=s.x}
   as MyCounterM R;
```

```
mc = {*MyCounterR,
       {state = {#x=0,#count=0,#old=0},
        methods = fix (myCounterClass [MyCounterR]) unit}}
      as Object MyCounterM;

sendget [MyCounterM]
  (sendreset [MyCounterM] (sendinc [MyCounterM] mc));
```

“我亲爱的 Watson，自己试着做一些分析，”他说，有一些不耐烦。“你知道我的方法。应用它们，对比较结果是会有帮助的。”

——A. Conan Doyle，四签名(1890)

如何完善它可以作为练习留给读者。这里我在做一件自己喜爱的工作，同时让数学家们少费点口舌。

——W. v. O. Quine (1987)

附录B 标 记 约 定

B.1 元变量名

文本中	在 ML 编码中	用法
p,q,r,s,t,u	s,t	项
x,y,z	x,y	项变量
v,w	v,w	值
nv	nv	数值
l,j,k	l	记录/变式字段
μ	store	存储
M,N,P,Q,S,T,U,V	tyS,tyT	类型
A,B,C	tyA,tyB	基础类型
Σ		存储类型化
X,Y,Z	tyX,tyY	类型变量
K,L	kK,kL	分类
σ		代换
Γ,Δ	ctx	上下文
$\mathcal{J}$		任意语句
$\mathcal{D}$		类型化导出
$\mathcal{C}$		子类型化导出
	fi	文件位置信息
i,j,k,l		数字下标

B.2 规则命名约定

前缀	用法
B-	大步求值
CT-	约束类型化
E-	求值
K-	类型化
M-	匹配
P-	模式类型化
Q-	类型等价
QR-	类型的并行归约
S-	子类型化
SA-	算法子类型化
T-	类型化
TA-	算法类型化
XA-	揭示

B.3 命名和下标的约定

全书中对元变量、数字下标和前提的选择基于以下原则：

1. 在语法定义中，元变量 t 用于表示所有的项，T 表示类型，v 表示值，等等。
2. 在类型化规则中，主要的项(其类型正在被计算的项)通常被称为 t，且它的子项被命名为 t_1，t_2，等等[偶尔地，(例如，在归约规则中)需要子项的子项的名称；对此我们使用 t_{11}，t_{12}，等等]。
3. 在求值规则中，整个正在被归约的项被称为 t，它被归约到的项被称为 t'。
4. 项 t 的类型被称为 T(同样，子项 t_1 的类型为 T_1，等等)。
5. 在描述和证明定理时也使用同样的约定，除非 t 有时被替换为 s(且 T 被替换为 S 或 R，等等)来避免在定义和定理之间的命名冲突。

存在这些规则不能同时满足的情况。在这样的情况中，最早的那个被赋予优先权(例如，在图 11.5 中，规则 T-Proj1 中的规则 4 没有完全遵守子项 t_1 的类型是 $T_1 \times T_2$)。在极少数的情况下，如果采用了某些规则会产生无法接受的可怕的或不可读的结果，则这些规则可完全忽略(例如图 11.7 中的记录投影规则 T-Proj)。

参考文献

Abadi, Martín. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5): 749–786, September 1999. Summary in *Theoretical Aspects of Computer Software (TACS), Sendai, Japan*, 1997; volume 1281 of Springer LNCS.

Abadi, Martín, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *ACM Symposium on Principles of Programming Languages (POPL), San Antonio, Texas*, pages 147–160, 1999.

Abadi, Martín and Luca Cardelli. On subtyping and matching. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 145–167, 1995.

Abadi, Martín and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

Abadi, Martín, Luca Cardelli, and Pierre-Louis Curien. Formal parametric polymorphism. *Theoretical Computer Science*, 121(1–2):9–58, 6 December 1993. Summary in *ACM Symposium on Principles of Programming Languages (POPL), Charleston, South Carolina*, 1993.

Abadi, Martín, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991a. Summary in *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*, 1990.

Abadi, Martín, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991b. Summary in *ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas*, 1989.

Abadi, Martín, Luca Cardelli, Benjamin Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, January 1995. Summary in *ACM SIGPLAN Workshop on ML and its Applications*, June 1992.

Abadi, Martín, Luca Cardelli, and Ramesh Viswanathan. An interpretation of objects and object types. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, pages 396–409, 1996.

Abadi, Martín and Marcelo P. Fiore. Syntactic considerations on recursive types. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science, LICS '96*, pages 242–252. IEEE Computer Society Press, Los Alamitos, CA, July 1996.

Abelson, Harold and Gerald Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, New York, 1985. Second edition, 1996.

Abramsky, Samson, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for pcf. *Information and Computation*, 163(2):409–470, December 2000.

Aczel, Peter. An introduction to inductive definitions. In Jon Barwise, editor, *Handbook of Mathematical Logic*, number 90 in Studies in Logic and the Foundations of Mathematics, pages 739–782. North Holland, 1977.

Aczel, Peter. *Non-Well-Founded Sets*. Stanford Center for the Study of Language and Information, 1988. CSLI Lecture Notes number 14.

Agesen, Ole, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the Java language. In *ACM Symposium on Object Oriented Programming: Sys-*

tems, Languages, and Applications (OOPSLA), pages 49–65, Atlanta, GA, October 1997.

Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, USA, 1986.

Aiken, Alexander and Edward L. Wimmers. Type inclusion constraints and type inference. In *ACM Symposium on Functional Programming Languages and Computer Architecture (FPCA)*, pages 31–41, 1993.

Amadio, Roberto M. and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993. Summary in *ACM Symposium on Principles of Programming Languages (POPL), Orlando, Florida*, pp. 104–118; also DEC/Compaq Systems Research Center Research Report number 62, August 1990.

Appel, Andrew W. *Modern Compiler Implementation in ML.* Cambridge University Press, 1998.

Appel, Andrew W. and Marcelo J. R. Gonçalves. Hash-consing garbage collection. Technical Report CS-TR-412-93, Princeton University, Computer Science Department, 1993.

Arbib, Michael and Ernest Manes. *Arrows, Structures, and Functors: The Categorical Imperative.* Academic Press, 1975.

Ariola, Zena M., Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. A call-by-need lambda calculus. In *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*, pages 233–246, January 1995.

Arnold, Ken and James Gosling. *The Java Programming Language.* Addison Wesley, 1996.

Arnold, Ken, Ann Wollrath, Bryan O'Sullivan, Robert Scheifler, and Jim Waldo. *The Jini specification.* Addison-Wesley, Reading, MA, USA, 1999.

Asperti, Andrea and Giuseppe Longo. *Categories, Types, and Structures: An Introduction to Category Theory for the Working Computer Scientist.* MIT Press, 1991.

Aspinall, David. Subtyping with singleton types. In *Computer Science Logic (CSL), Kazimierz, Poland*, pages 1–15. Springer-Verlag, 1994.

Aspinall, David and Adriana Compagnoni. Subtyping dependent types. *Information and Computation*, 266(1–2):273–309, September 2001. Preliminary version in *IEEE Symposium on Logic in Computer Science (LICS)*, 1996.

Astesiano, Egidio. Inductive and operational semantics. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, IFIP State-of-the-Art Reports, pages 51–136. Springer-Verlag, 1991.

Augustsson, Lennart. A compiler for Lazy ML. In *ACM Symposium on Lisp and Functional Programming (LFP), Austin, Texas*, pages 218–227, August 1984.

Augustsson, Lennart. Cayenne — a language with dependent types. In *International Conference on Functional Programming (ICFP), Baltimore, Maryland, USA*, pages 239–250, 1998.

Baader, Franz and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

Baader, Franz and Jörg Siekmann. Unification theory. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and*

Logic Programming, volume 2, Deduction Methodologies, pages 41–125. Oxford University Press, Oxford, UK, 1994.

Backus, John. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978. Reproduced in *Selected Reprints on Dataflow and Reduction Architectures*, ed. S. S. Thakkar, IEEE, 1987, pp. 215–243, and in *ACM Turing Award Lectures: The First Twenty Years*, ACM Press, 1987, pp. 63–130.

Backus, John. The history of Fortran I, II, and III. In Wexelblat, editor, *History of Programming Languages*, pages 25–45. Academic Press, 1981.

Bainbridge, E. Stewart, Peter J. Freyd, Andre Scedrov, and Philip J. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70(1):35–64, 1990. Corrigendum in *TCS* 71(3), 431.

Baldan, Paolo, Giorgio Ghelli, and Alessandra Raffaetà. Basic theory of F-bounded quantification. *Information and Computation*, 153(1):173–237, 1999.

Barendregt, Henk P. *The Lambda Calculus.* North Holland, revised edition, 1984.

Barendregt, Henk P. Functional programming and lambda calculus. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, chapter 7, pages 321–364. Elsevier / MIT Press, 1990.

Barendregt, Henk P. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991.

Barendregt, Henk P. Lambda calculi with types. In Abramsky, Gabbay, and Maibaum, editors, *Handbook of Logic in Computer Science*, volume II. Oxford University Press, 1992.

Barras, Bruno, Samuel Boutin, Cristina Cornes, Judicael Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saibi, and Benjamin Werner. The Coq proof assistant reference manual : Version 6.1. Technical Report RT-0203, Inria (Institut National de Recherche en Informatique et en Automatique), France, 1997.

Barwise, Jon and Lawrence Moss. *Vicious Circles: On the Mathematics of Non-wellfounded Phenomena.* Cambridge University Press, 1996.

Berardi, Stefano. Towards a mathematical analysis of the Coquand-Huet calculus of constructions and the other systems in Barendregt's cube. Technical report, Department of Computer Science, CMU, and Dipartimento Matematica, Universita di Torino, 1988.

Berger, Ulrich. Program extraction from normalization proofs. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, pages 91–106, Utrecht, The Netherlands, March 1993. Springer-Verlag.

Berger, Ulrich and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In Gilles Kahn, editor, *IEEE Symposium on Logic in Computer Science (LICS)*, pages 203–211, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

Birtwistle, Graham M., Ole-Johan Dahl, Bjorn Myhrhaug, and Kristen Nygaard. *Simula Begin.* Studentlitteratur (Lund, Sweden), Bratt Institut fuer neues Lernen (Goch, FRG), Chartwell-Bratt Ltd (Kent, England), 1979.

Bobrow, Daniel G., Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp Object System specification X3J13 document 88-002R. *SIGPLAN Notices*, 23, 1988.

Boehm, Hans-J. Partial polymorphic type inference is undecidable. In *26th Annual Symposium on Foundations of Computer Science*, pages 339-345. IEEE, October 1985.

Boehm, Hans-J. Type inference in the presence of type abstraction. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Portland, Oregon*, pages 192-206, June 1989.

Böhm, Corrado and Alessandro Berarducci. Automatic synthesis of typed Λ-programs on term algebras. *Theoretical Computer Science*, 39(2-3):135-154, August 1985.

Bono, Viviana and Kathleen Fisher. An imperative first-order calculus with object extension. In *European Conference on Object-Oriented Programming (ECOOP)*, 1998.

Bono, Viviana, Amit J. Patel, and Vitaly Shmatikov. A core calculus of classes and mixins. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 1628 of *Lecture Notes in Computer Science*, pages 43-66. Springer-Verlag, June 1999a.

Bono, Viviana, Amit J. Patel, Vitaly Shmatikov, and John C. Mitchell. A core calculus of classes and objects. In *Fifteenth Conference on the Mathematical Foundations of Programming Semantics*, April 1999b.

Bracha, Gilad, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In Craig Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, ACM SIGPLAN Notices volume 33 number 10, pages 183-200, Vancouver, BC, October 1998.

Braithwaite, Richard B. *The Foundations of Mathematics: Collected Papers of Frank P. Ramsey*. Routledge and Kegan Paul, London, 1931.

Brandt, Michael and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. In Roger Hindley, editor, *Proc. 3d Int'l Conf. on Typed Lambda Calculi and Applications (TLCA), Nancy, France, April 2-4, 1997*, volume 1210 of *Lecture Notes in Computer Science (LNCS)*, pages 63-81. Springer-Verlag, April 1997. Full version in Fundamenta Informaticae, Vol. 33, pp. 309-338, 1998.

Breazu-Tannen, Val, Thierry Coquand, Carl Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172-221, 1991. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

Bruce, Kim B. The equivalence of two semantic definitions for inheritance in object-oriented languages. In *Proceedings of Mathematical Foundations of Programming Semantics*, Pittsburgh, PA, March 1991.

Bruce, Kim B. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2), April 1994. Summary in *ACM Symposium on Principles of Programming Languages (POPL), Charleston, South Carolina*, under the title "Safe type checking in a statically typed object-oriented programming language".

Bruce, Kim B. *Foundations of Object-Oriented Languages: Types and Semantics*. MIT Press, 2002.

Bruce, Kim B., Luca Cardelli, Giuseppe Castagna, the Hopkins Objects Group (Jonathan Eifrig, Scott Smith, Valery Trifonov), Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1996.

Bruce, Kim B., Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 155(1/2):108–133, November 1999. Special issue of papers from *Theoretical Aspects of Computer Software (TACS 1997)*. An earlier version appeared as an invited lecture in the Third International Workshop on Foundations of Object Oriented Languages (FOOL 3), July 1996.

Bruce, Kim B. and Giuseppe Longo. A modest model of records, inheritance, and bounded quantification. *Information and Computation*, 87:196–240, 1990. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994). An earlier version appeared in the proceedings of the IEEE Symposium on Logic in Computer Science, 1988.

Bruce, Kim B. and John Mitchell. PER models of subtyping, recursive types and higher-order polymorphism. In *ACM Symposium on Principles of Programming Languages (POPL), Albuquerque, New Mexico*, January 1992.

Bruce, Kim B., Leaf Petersen, and Adrian Fiech. Subtyping is not a good "match" for object-oriented languages. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 104–127. Springer-Verlag, 1997.

Buneman, Peter and Benjamin Pierce. Union types for semistructured data. In *Internet Programming Languages*. Springer-Verlag, September 1998. Proceedings of the International Database Programming Languages Workshop. LNCS 1686.

Burstall, Rod and Butler Lampson. A kernel language for abstract data types and modules. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 1–50. Springer-Verlag, 1984.

Burstall, Rod M. Proving properties of programs by structural induction. *The Computer Journal*, 12(1):41–48, 1969.

Canning, Peter, William Cook, Walt Hill, and Walter Olthoff. Interfaces for strongly-typed object-oriented programming. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 457–467, 1989a.

Canning, Peter, William Cook, Walter Hill, Walter Olthoff, and John Mitchell. F-bounded quantification for object-oriented programming. In *ACM Symposium on Functional Programming Languages and Computer Architecture (FPCA)*, pages 273–280, September 1989b.

Canning, Peter, Walt Hill, and Walter Olthoff. A kernel language for object-oriented programming. Technical Report STL-88-21, Hewlett-Packard Labs, 1988.

Cardelli, Luca. A semantics of multiple inheritance. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67. Springer-Verlag, 1984. Full version in *Information and Computation*, 76(2/3):138–164, 1988.

Cardelli, Luca. Amber. In Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and Functional Programming Languages*, pages 21–47. Springer-Verlag, 1986. Lecture Notes in Computer Science No. 242.

Cardelli, Luca. Basic polymorphic typechecking. *Science of Computer Programming*, 8 (2):147-172, April 1987. An earlier version appeared in the *Polymorphism Newsletter*, January, 1985.

Cardelli, Luca. Structural subtyping and the notion of power type. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, pages 70-79, January 1988a.

Cardelli, Luca. Typechecking dependent types and subtypes. In M. Boscarol, L. Carlucci Aiello, and G. Levi, editors, *Foundations of Logic and Functional Programming, Workshop Proceedings, Trento, Italy, (Dec. 1986)*, volume 306 of *Lecture Notes in Computer Science*, pages 45-57. Springer-Verlag, 1988b.

Cardelli, Luca. Notes about $F^{\omega}_{<:}$. Unpublished manuscript, October 1990.

Cardelli, Luca. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*. Springer-Verlag, 1991. An earlier version appeared as DEC/Compaq Systems Research Center Research Report #45, February 1989.

Cardelli, Luca. Extensible records in a pure calculus of subtyping. Research report 81, DEC/Compaq Systems Research Center, January 1992. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

Cardelli, Luca. An implementation of $F_{<:}$. Research report 97, DEC/Compaq Systems Research Center, February 1993.

Cardelli, Luca. Type systems. In Allen B. Tucker, editor, *Handbook of Computer Science and Engineering*. CRC Press, 1996.

Cardelli, Luca, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report (revised). Research report 52, DEC/Compaq Systems Research Center, November 1989.

Cardelli, Luca and Xavier Leroy. Abstract types and the dot notation. In *Proceedings of the IFIP TC2 Working Conference on Programming Concepts and Methods*. North Holland, 1990. Also appeared as DEC/Compaq SRC technical report 56.

Cardelli, Luca and Giuseppe Longo. A semantic basis for Quest. *Journal of Functional Programming*, 1(4):417-458, October 1991. Summary in ACM Conference on Lisp and Functional Programming, June 1990. Also available as DEC/Compaq SRC Research Report 55, Feb. 1990.

Cardelli, Luca, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of System F with subtyping. *Information and Computation*, 109(1-2):4-56, 1994. Summary in TACS '91 (Sendai, Japan, pp. 750-770).

Cardelli, Luca and John Mitchell. Operations on records. *Mathematical Structures in Computer Science*, 1:3-48, 1991. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994); available as DEC/Compaq Systems Research Center Research Report #48, August, 1989, and in the proceedings of MFPS '89, Springer LNCS volume 442.

Cardelli, Luca and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471-522, December 1985.

Cardone, Felice. Relational semantics for recursive types and bounded quantification. In *Proceedings of the Sixteenth International Colloquium on Automata, Languages, and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 164-178, Stresa, Italy, July 1989. Springer-Verlag.

Cardone, Felice and Mario Coppo. Type inference with recursive types: Syntax and semantics. *Information and Computation*, 92(1):48–80, 1991.

Cartwright, Robert and Guy L. Steele, Jr. Compatible genericity with run-time types for the Java programming language. In Craig Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA), Vancouver, British Columbia*, SIGPLAN Notices 33(10), pages 201–215. ACM, October 1998.

Castagna, Giuseppe. *Object-Oriented Programming: A Unified Foundation*. Springer-Verlag, 1997.

Castagna, Giuseppe, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, 15 February 1995. preliminary version in LISP and Functional Programming, July 1992 (pp. 182–192), and as Rapport de Recherche LIENS-92-4, Ecole Normale Supérieure, Paris.

Chambers, Craig. Object-oriented multi-methods in Cecil. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 33–56, 1992.

Chambers, Craig. The Cecil language: Specification and rationale. Technical report, University of Washington, March 1993.

Chambers, Craig and Gary Leavens. Type-checking and modules for multi-methods. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, October 1994. SIGPLAN Notices 29(10).

Chen, Gang and Giuseppe Longo. Subtyping parametric and dependent types. In Kamareddine et al., editor, *Type Theory and Term Rewriting*, September 1996. Invited lecture.

Chirimar, Jawahar, Carl A. Gunter, and Jon G. Riecke. Reference counting as a computational interpretation of linear logic. *Journal of Functional Programming*, 6(2): 195–244, March 1996.

Church, Alonzo. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:354–363, 1936.

Church, Alonzo. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

Church, Alonzo. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.

Clement, Dominique, Joelle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ML. In *ACM Conference on LISP and Functional Programming*, pages 13–27, 1986.

Clinger, William, Daniel P. Friedman, and Mitchell Wand. A scheme for a higher-level semantic algebra. In John Reynolds and Maurice Nivat, editors, *Algebraic Methods in Semantics*, pages 237–250. Cambridge University Press, 1985.

Colazzo, Dario and Giorgio Ghelli. Subtyping recursive types in Kernel Fun. In *14th Symposium on Logic in Computer Science (LICS'99)*, pages 137–146. IEEE, July 1999.

Compagnoni, Adriana and Healfdene Goguen. Decidability of higher-order subtyping via logical relations, December 1997a. Manuscript, available at `ftp://www.dcs.ed.ac.uk/pub/hhg/hosdec.ps.gz`.

Compagnoni, Adriana and Healfdene Goguen. Typed operational semantics for higher order subtyping. Technical Report ECS-LFCS-97-361, University of Edinburgh, July 1997b.

Compagnoni, Adriana B. Decidability of higher-order subtyping with intersection types. In *Computer Science Logic*, September 1994. Kazimierz, Poland. Springer *Lecture Notes in Computer Science* 933, June 1995. Also available as University of Edinburgh, LFCS technical report ECS-LFCS-94-281, titled "Subtyping in $F^{\omega}_{\wedge}$ is decidable".

Compagnoni, Adriana B. and Benjamin C. Pierce. Intersection types and multiple inheritance. *Mathematical Structures in Computer Science*, 6(5):469–501, October 1996. Preliminary version available as University of Edinburgh technical report ECS-LFCS-93-275 and Catholic University Nijmegen computer science technical report 93-18, Aug. 1993, under the title "Multiple Inheritance via Intersection Types".

Constable, Robert L. Types in computer science, philosophy, and logic. In Samuel R. Buss, editor, *Handbook of Proof Theory*, volume 137 of *Studies in logic and the foundations of mathematics*, pages 683–786. Elsevier, 1998.

Constable et al., Robert L. *Implementing Mathematics with the NuPRL Proof Development System*. Prentice-Hall, Englewood Cliffs, NJ, 1986.

Cook, William. Object-oriented programming versus abstract data types. In J. W. de Bakker et al., editors, *Foundations of Object-Oriented Languages*, volume 489 of *Lecture Notes in Computer Science*, pages 151–178. Springer-Verlag, 1991.

Cook, William and Jens Palsberg. A denotational semantics of inheritance and its correctness. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 433–444, 1989.

Cook, William R. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.

Cook, William R., Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*, pages 125–135, January 1990. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

Coppo, Mario and Mariangiola Dezani-Ciancaglini. A new type-assignment for λ-terms. *Archiv Math. Logik*, 19:139–156, 1978.

Coppo, Mario, Mariangiola Dezani-Ciancaglini, and Patrick Sallé. Functional characterization of some semantic equalities inside λ-calculus. In Hermann A. Maurer, editor, *Proceedings of the 6th Colloquium on Automata, Languages and Programming*, volume 71 of *LNCS*, pages 133–146, Graz, Austria, July 1979. Springer.

Coquand, Thierry. *Une Théorie des Constructions*. PhD thesis, University Paris VII, January 1985.

Coquand, Thierry and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

Courcelle, Bruno. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.

Cousineau, Guy and Michel Mauny. *The Functional Approach to Programming*. Cambridge University Press, 1998.

Crary, Karl. Sound and complete elimination of singleton kinds. Technical Report CMU-CS-00-104, Carnegie Mellon University, School of Computer Science, January 2000.

Crary, Karl, Robert Harper, and Derek Dreyer. A type system for higher-order modules. In *ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, 2002.

Crary, Karl, Robert Harper, and Sidd Puri. What is a recursive module? In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 50-63, May 1999.

Crary, Karl, Stephanie Weirich, and J. Gregory Morrisett. Intensional polymorphism in type-erasure semantics. In *International Conference on Functional Programming (ICFP), Baltimore, Maryland, USA*, pages 301-312, 1998.

Crole, Roy. *Categories for Types.* Cambridge University Press, 1994.

Curien, Pierre-Louis and Giorgio Ghelli. Subtyping + extensionality: Confluence of $\beta\eta$-reductions in $F_{\leq}$. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software (Sendai, Japan)*, number 526 in Lecture Notes in Computer Science, pages 731-749. Springer-Verlag, September 1991.

Curien, Pierre-Louis and Giorgio Ghelli. Coherence of subsumption: Minimum typing and type-checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2:55-91, 1992. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

Curry, Haskell B. and Robert Feys. *Combinatory Logic*, volume 1. North Holland, 1958. Second edition, 1968.

Damas, Luis and Robin Milner. Principal type schemes for functional programs. In *ACM Symposium on Principles of Programming Languages (POPL), Albuquerque, New Mexico*, pages 207-212, 1982.

Danvy, Olivier. Type-directed partial evaluation. In John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors, *Partial Evaluation - Practice and Theory; Proceedings of the 1998 DIKU Summer School*, number 1706 in Lecture Notes in Computer Science, pages 367-411, Copenhagen, Denmark, July 1998. Springer-Verlag.

Davey, Brian A. and Hilary A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, 1990.

Davies, Rowan. A refinement-type checker for Standard ML. In *International Conference on Algebraic Methodology and Software Technology*, volume 1349 of *Lecture Notes in Computer Science.* Springer-Verlag, 1997.

Davies, Rowan and Frank Pfenning. A modal analysis of staged computation. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, pages 258-270, 1996.

de Bruijn, Nicolas G. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381-392, 1972.

de Bruijn, Nicolas G. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589-606. Academic Press, 1980.

De Millo, Richard A., Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271-280, May 1979.

An earlier version appeared in *ACM Symposium on Principles of Programming Languages (POPL), Los Angeles, California*, 1977 pp. 206–214.

Detlefs, David L., K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report 159, Compaq Systems Research Center (SRC), 1998. Also see `http://research.compaq.com/SRC/esc/overview.html`.

Donahue, James and Alan Demers. Data types are values. *ACM Transactions on Programming Languages and Systems*, 7(3):426–445, July 1985.

Dowek, Gilles, Thérèse Hardin, Claude Kirchner, and Frank Pfenning. Unification via explicit substitutions: The case of higher-order patterns. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 259–273, Bonn, Germany, September 1996. MIT Press.

Drossopoulou, Sophia, Susan Eisenbach, and Sarfraz Khurshid. Is the Java Type System Sound? *Theory and Practice of Object Systems*, 7(1):3–24, 1999. Summary in European Conference on Object-Oriented Programming (ECOOP), 1997.

Duggan, Dominic and Adriana Compagnoni. Subtyping for object type constructors. In *Workshop on Foundations of Object-Oriented Languages (FOOL), informal proceedings*, January 1999.

Dybvig, R. Kent. *The Scheme Programming Language*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, second edition, 1996. Available electronically at `http://www.scheme.com/tspl2d/`.

Eidorff, Peter, Fritz Henglein, Christian Mossin, Henning Niss, Morten Heine B. Sørensen, and Mads Tofte. AnnoDomini in practice: A type-theoretic approach to the Year 2000 problem. In Jean-Yves Girard, editor, *Proc. Symposium on Typed Lambda Calculus and Applications (TLCA)*, volume 1581 of *Lecture Notes in Computer Science*, pages 6–13, L'Aquila, Italy, April 1999. Springer-Verlag.

Eifrig, Jonathan, Scott Smith, and Valery Trifonov. Type inference for recursively constrained types and its application to OOP. In *Proceedings of the 1995 Mathematical Foundations of Programming Semantics Conference*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995.

Feinberg, Neal, Sonya E. Keene, Robert O. Mathews, and P. Tucker Withington. *The Dylan Programming Book*. Addison-Wesley Longman, Reading, Mass., 1997.

Felleisen, Matthias and Daniel P. Friedman. *A Little Java, A Few Patterns*. MIT Press, Cambridge, Massachusetts, 1998.

Felty, Amy, Elsa Gunter, John Hannan, Dale Miller, Gopalan Nadathur, and Andre Scedrov. Lambda prolog: An extended logic programming language. In E. Lusk; R. Overbeek, editor, *Proceedings on the 9th International Conference on Automated Deduction*, volume 310 of *LNCS*, pages 754–755, Berlin, May 1988. Springer.

Filinski, Andrzej. A semantic account of type-directed partial evaluation. In Gopalan Nadathur, editor, *Proceedings of the International Conference on Principles and Practice of Declarative Programming*, number 1702 in Lecture Notes in Computer Science, pages 378–395, Paris, France, September 1999. Springer-Verlag. Extended version available as technical report BRICS RS-99-17.

Filinski, Andrzej. Normalization by evaluation for the computational lambda-calculus. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001*, number 2044 in Lecture Notes in Computer Science, pages 151–165, Krakow, Poland, May 2001. Springer-Verlag.

Fisher, Kathleen. Classes = objects + data abstraction. In Kim Bruce and Giuseppe Longo, editors, *Workshop on Foundations of Object-Oriented Languages (FOOL), informal proceedings*, July 1996a. Invited talk. Also available as Stanford University Technical Note STAN-CS-TN-96-31.

Fisher, Kathleen. *Type Systems for object-oriented programming languages.* PhD thesis, Stanford University, 1996b. STAN-CS-TR-98-1602.

Fisher, Kathleen, Furio Honsell, and John C. Mitchell. A lambda calculus of objects and method specialization. *Nordic J. Computing* (formerly *BIT*), 1:3-37, 1994. Summary in *Proc. IEEE Symp. on Logic in Computer Science*, 1993, 26-38.

Fisher, Kathleen and John Mitchell. The development of type systems for object-oriented languages. *Theory and Practice of Object Systems*, 1(3):189-220, 1996.

Fisher, Kathleen and John C. Mitchell. On the relationship between classes, objects, and data abstraction. *Theory and Practice of Object Systems*, 4(1):3-25, 1998.

Fisher, Kathleen and John H. Reppy. The design of a class mechanism for Moby. In *SIGPLAN Conference on Programming Language Design and Implementation (PDLI)*, pages 37-49, 1999.

Flanagan, Cormac and Matthias Felleisen. Componential set-based analysis. *ACM SIGPLAN Notices*, 32(5):235-248, May 1997.

Flatt, Matthew and Matthias Felleisen. Units: Cool modules for HOT languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada*, pages 236-248, 1998.

Flatt, Matthew, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, January 1998a.

Flatt, Matthew, Shriram Krishnamurthi, and Matthias Felleisen. A programmer's reduction semantics for classes and mixins. Technical Report TR97-293, Computer Science Department, Rice University, February 1998b. Corrected June, 1999.

Freeman, Tim and Frank Pfenning. Refinement types for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario*, June 1991.

Frege, Gottlob. *Begriffschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens.* Halle: L. Nebert, 1879. Available in several translations.

Friedman, Daniel P. and Matthias Felleisen. *The Little Schemer.* MIT Press, 1996.

Friedman, Daniel P., Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages.* McGraw-Hill Book Co., New York, N.Y., second edition, 2001.

Friedman, Harvey. Equality between functionals. In Rohit Parikh, editor, *Logic Colloquium*, volume 453 of *Lecture Notes in Mathematics*, pages 22-37, Berlin, 1975. Springer-Verlag.

Gallier, Jean. On Girard's "Candidats de reductibilité". In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, number 31 in APIC Studies in Data Processing, pages 123-203. Academic Press, 1990.

Gallier, Jean. Constructive logics. Part I: A tutorial on proof systems and typed λ-calculi. *Theoretical Computer Science*, 110(2):249-339, March 1993.

Gandy, Robin O. The simple theory of types. In *Logic Colloquium 76*, volume 87 of *Studies in Logic and the Foundations of Mathematics*, pages 173-181. North Holland, 1976.

Gapeyev, Vladimir, Michael Levin, and Benjamin Pierce. Recursive subtyping revealed.

In *International Conference on Functional Programming (ICFP), Montreal, Canada*, 2000. To appear in *Journal of Functional Programming*.

Garrigue, Jaques and Hassan Aït-Kaci. The typed polymorphic label-selective lambda-calculus.. In *ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, pages 35–47, 1994.

Garrigue, Jaques and Didier Rémy. Extending ML with semi-explicit polymorphism. In Martín Abadi and Takayasu Ito, editors, *International Symposium on Theoretical Aspects of Computer Software (TACS), Sendai, Japan*, pages 20–46. Springer-Verlag, September 1997.

Ghelli, Giorgio. *Proof Theoretic Studies about a Minimal Type System Integrating Inclusion and Parametric Polymorphism*. PhD thesis, Università di Pisa, March 1990. Technical report TD-6/90, Dipartimento di Informatica, Università di Pisa.

Ghelli, Giorgio. Recursive types are not conservative over $F_{\leq}$. In M. Bezen and J.F. Groote, editors, *Typed Lambda Calculi and Applications (TLCA), Utrecht, The Netherlands*, number 664 in Lecture Notes in Computer Science, pages 146–162, Berlin, March 1993. Springer-Verlag.

Ghelli, Giorgio. Divergence of $F_{\leq}$ type checking. *Theoretical Computer Science*, 139 (1,2):131–162, 1995.

Ghelli, Giorgio. Termination of system F-bounded: A complete proof. *Information and Computation*, 139(1):39–56, 1997.

Ghelli, Giorgio and Benjamin Pierce. Bounded existentials and minimal typing. *Theoretical Computer Science*, 193:75–96, 1998.

Gifford, David, Pierre Jouvelot, John Lucassen, and Mark Sheldon. FX-87 Reference Manual. Technical Report MIT/LCS/TR-407, Massachusetts Institute of Technology, Laboratory for Computer Science, September 1987.

Girard, Jean-Yves. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse d'état, Université Paris VII, 1972. Summary in *Proceedings of the Second Scandinavian Logic Symposium* (J.E. Fenstad, editor), North-Holland, 1971 (pp. 63–92).

Girard, Jean-Yves. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

Girard, Jean-Yves, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

Glew, Neal. Type dispatch for named hierarchical types. In *International Conference on Functional Programming (ICFP), Paris, France*, pages 172–182, 1999.

Gordon, Andrew. A tutorial on co-induction and functional programming. In *Functional Programming, Glasgow 1994*, pages 78–95. Springer Workshops in Computing, 1995.

Gordon, Michael J. Adding eval to ML. Manuscript, circa 1980.

Gordon, Michael J., Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*. Springer-Verlag LNCS 78, 1979.

Goto, Eiichi. Monocopy and associative algorithms in extended Lisp. Technical Report TR 74-03, University of Tokyo, May 1974.

Goubault-Larrecq, Jean and Ian Mackie. *Proof Theory and Automated Deduction (Applied Logic Series, V. 6)*. Kluwer, 1997.

Grattan-Guinness, Ivor. *The search for mathematical roots, 1870-1940: Logics, set theories and the foundations of mathematics from Cantor through Russell to Gödel*.

Princeton University Press, 2001.

Gries, David, editor. *Programming Methodology.* Springer-Verlag, New York, 1978.

Gunter, Carl A. *Semantics of Programming Languages: Structures and Techniques.* MIT Press, 1992.

Gunter, Carl A. and John C. Mitchell. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design.* MIT Press, 1994.

Hall, Cordelia V., Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems,* 18 (2):109–138, March 1996.

Halmos, Paul R. *Naive Set Theory.* Springer, New York, 1987.

Harper, Robert. A simplified account of polymorphic references. *Information Processing Letters,* 51(4):201–206, August 1994. See also (Harper, 1996).

Harper, Robert. A note on: "A simplified account of polymorphic references" [Inform. Process. Lett. **51** (1994), no. 4, 201–206; MR 95f:68142]. *Information Processing Letters,* 57(1):15–16, January 1996. See (Harper, 1994).

Harper, Robert, Bruce Duba, and David MacQueen. First-class continuations in ML. *Journal of Functional Programming,* 3(4), October 1993. Short version in POPL '91.

Harper, Robert, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM,* 40(1):143–184, 1992. Summary in LICS'87.

Harper, Robert and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon,* pages 123–137, January 1994.

Harper, Robert, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California,* pages 341–354, January 1990.

Harper, Robert and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California,* pages 130–141, 1995.

Harper, Robert and Benjamin Pierce. A record calculus based on symmetric concatenation. In *ACM Symposium on Principles of Programming Languages (POPL), Orlando, Florida,* pages 131–142, January 1991. Extended version available as Carnegie Mellon Technical Report CMU-CS-90-157.

Harper, Robert and Christopher Stone. A type-theoretic interpretation of Standard ML. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner.* MIT Press, 2000.

Hasegawa, Ryu. Parametricity of extensionally collapsed term models of polymorphism and their categorical properties. In Takayasu Ito and Albert Meyer, editors, *Theoretical Aspects of Computer Software (TACS), Sendai, Japan,* 1991.

Hayashi, Susumu. Singleton, union and intersection types for program extraction. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software (Sendai, Japan),* number 526 in Lecture Notes in Computer Science, pages 701–730. Springer-Verlag, September 1991. Full version in *Information and Computation,* 109(1/2):174–210, 1994.

Henglein, Fritz. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems,* 15(2):253–289, 1993.

Henglein, Fritz. Dynamic typing: syntax and proof theory. *Science of Computer Programming,* 22(3):197–230, June 1994. Selected papers of the Fourth European

Symposium on Programming (Rennes, 1992).

Henglein, Fritz and Harry G. Mairson. The complexity of type inference for higher-order typed lambda-calculi. In *ACM Symposium on Principles of Programming Languages (POPL), Orlando, Florida*, pages 119–130, January 1991.

Hennessy, Matthew. *A Semantics of Programming Languages: An Elementary Introduction Using Operational Semantics.* John Wiley and Sons, 1990. Currently out of print; available from `http://www.cogs.susx.ac.uk/users/matthewh/semnotes.ps.gz`.

Hennessy, Matthew and James Riely. Resource access control in systems of mobile agents. In Uwe Nestmann and Benjamin C. Pierce, editors, *HLCL '98: High-Level Concurrent Languages (Nice, France, September 12, 1998)*, volume 16.3 of *ENTCS*, pages 3–17. Elsevier Science Publishers, 1998. Full version available as CogSci Report 2/98, University of Sussex, Brighton.

Hindley, J. Roger. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

Hindley, J. Roger. Types with intersection, an introduction. *Formal Aspects of Computing*, 4:470–486, 1992.

Hindley, J. Roger. *Basic Simple Type Theory*, volume 42 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, Cambridge, 1997.

Hindley, J. Roger and Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus*, volume 1 of *London Mathematical Society Student Texts.* Cambridge University Press, 1986.

Hoang, My, John Mitchell, and Ramesh Viswanathan. Standard ML-NJ weak polymorphism and imperative constructs. In *Proceedings, Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 15–25. IEEE Computer Society Press, 1993.

Hodas, J. S. Lolli: An extension of λProlog with linear context management. In D. Miller, editor, *Workshop on the λProlog Programming Language*, pages 159–168, Philadelphia, Pennsylvania, August 1992.

Hofmann, Martin. Syntax and semantics of dependent types. In *Semantics and Logic of Computation.* Cambridge University Press, 1997.

Hofmann, Martin and Benjamin Pierce. Positive subtyping. In *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*, pages 186–197, January 1995a. Full version in *Information and Computation*, volume 126, number 1, April 1996. Also available as University of Edinburgh technical report ECS-LFCS-94-303, September 1994.

Hofmann, Martin and Benjamin Pierce. A unifying type-theoretic framework for objects. *Journal of Functional Programming*, 5(4):593–635, October 1995b. Previous versions appeared in the Symposium on Theoretical Aspects of Computer Science, 1994, (pages 251–262) and, under the title "An Abstract View of Objects and Subtyping (Preliminary Report)," as University of Edinburgh, LFCS technical report ECS-LFCS-92-226, 1992.

Hofmann, Martin and Benjamin C. Pierce. Type destructors. In Didier Rémy, editor, *Informal proceedings of the Fourth International Workshop on Foundations of Object-Oriented Languages (FOOL)*, January 1998. Full version to appear in *Information and Computation.*

Hook, J.G. Understanding Russell – a first attempt. In *Proc. Int. Symp. on Semantics of*

Data Types, Sophia-Antipolis (France), Springer LNCS 173, pages 69–85. Springer-Verlag, 1984.

Hopcroft, John E. and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

Hosoya, Haruo and Benjamin Pierce. Regular expression pattern matching. In *ACM Symposium on Principles of Programming Languages (POPL), London, England*, 2001.

Hosoya, Haruo and Benjamin C. Pierce. How good is local type inference? Technical Report MS-CIS-99-17, University of Pennsylvania, June 1999.

Hosoya, Haruo and Benjamin C. Pierce. XDuce: A typed XML processing language (preliminary report). In *International Workshop on the Web and Databases (WebDB)*, May 2000.

Hosoya, Haruo, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2001. To appear; short version in ICFP 2000.

Howard, William A. Hereditarily majorizable functionals of finite type. In Anne Sjerp Troelstra, editor, *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, volume 344 of *Lecture Notes in Mathematics*, pages 454–461. Springer-Verlag, Berlin, 1973. Appendix.

Howard, William A. The formulas-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, New York, 1980. Reprint of 1969 article.

Howe, Douglas. *Automating Reasoning in an Implementation of Constructive Type Theory*. PhD thesis, Cornell University, 1988.

Hudak, Paul, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzman, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. Report on the programming language Haskell, version 1.2. *SIGPLAN Notices*, 27(5), May 1992.

Huet, Gérard. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

Huet, Gérard. *Résolution d'equations dans les langages d'ordre 1,2, ...,ω*. Thèse de Doctorat d'Etat, Université de Paris 7 (France), 1976.

Huet, Gérard, editor. *Logical Foundations of Functional Programming*. University of Texas at Austin Year of Programming Series. Addison-Wesley, 1990.

Hyland, J. Martin E. and C.-H. Luke Ong. On full abstraction for PCF: I, II, and III. *Information and Computation*, 163(2):285–408, December 2000.

Igarashi, Atsushi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, October 1999. Full version to appear in ACM Transactions on Programming Languages and Systems (TOPLAS), 2001.

Igarashi, Atsushi and Benjamin C. Pierce. On inner classes. In *European Conference on Object-Oriented Programming (ECOOP)*, 2000. Also in informal proceedings of the Seventh International Workshop on Foundations of Object-Oriented Languages (FOOL). To appear in *Information and Computation*.

Igarashi, Atsushi, Benjamin C. Pierce, and Philip Wadler. A recipe for raw types. In *Workshop on Foundations of Object-Oriented Languages (FOOL)*, 2001.

Ishtiaq, Samin and Peter O'Hearn. Bi as an assertion language for mutable data structures. In *ACM Symposium on Principles of Programming Languages (POPL), London, England*, 2001.

Jacobs, Bart. *Categorical Logic and Type Theory*. Number 141 in Studies in Logic and the Foundations of Mathematics. North Holland, Elsevier, 1999.

Jagannathan, Suresh and Andrew Wright. Effective flow analysis for avoiding runtime checks. In *Proceedings of the Second International Static Analysis Symposium*, volume 983 of *LNCS*, pages 207–224. Springer-Verlag, 1995.

Jay, C. Barry and Milan Sekanina. Shape checking of array programs. In *Computing: The Australasian Theory Seminar (Proceedings)*, volume 19 of *Australian Computer Science Communications*, pages 113–121, 1997.

Jim, Trevor. Rank-2 type systems and recursive definitions. Technical Report MIT/LCS/TM-531, Massachusetts Institute of Technology, Laboratory for Computer Science, November 1995.

Jim, Trevor. What are principal typings and what are they good for? In ACM, editor, *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, pages 42–53, 1996.

Jim, Trevor and Jens Palsberg. Type inference in systems of recursive types with subtyping. Manuscript, 1999.

Jones, Mark P. ML typing, explicit polymorphism, and qualified types, 1994a.

Jones, Mark P. *Qualified Types: Theory and Practice*. Cambridge University Press, 1994b.

Jones, Richard and Rafael D. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.

Jouvelot, Pierre and David Gifford. Algebraic reconstruction of types and effects. In *ACM Symposium on Principles of Programming Languages (POPL), Orlando, Florida*, pages 303–310, January 1991.

Jutting, L.S. van Benthem, James McKinna, and Robert Pollack. Checking algorithms for Pure Type Systems. In Henk Barendregt and Tobias Nipkow, editors, *Proceedings of the International Workshop on Types for Proofs and Programs*, pages 19–61, Nijmegen, The Netherlands, May 1994. Springer-Verlag LNCS 806.

Kaes, Stefan. Parametric overloading in polymorphic programming languages. In H. Ganzinger, editor, *Proceedings of the European Symposium on Programming*, volume 300 of *Lecture Notes in Computer Science*, pages 131–144. Springer-Verlag, 1988.

Kahn, Gilles. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1987.

Kamin, Samuel N. Inheritance in Smalltalk-80: A denotational definition. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, pages 80–87, January 1988.

Kamin, Samuel N. and Uday S. Reddy. Two semantic models of object-oriented languages. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 464–

495. MIT Press, 1994.

Katiyar, Dinesh, David Luckham, and John Mitchell. A type system for prototyping languages. In *ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, pages 138–150, January 1994.

Katiyar, Dinesh and Sriram Sankar. Completely bounded quantification is decidable. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, June 1992.

Kelsey, Richard, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998.

Kennedy, Andrew. Dimension types. In Donald Sannella, editor, *Programming Languages and Systems—ESOP'94, 5th European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 348–362, Edinburgh, U.K., 11–13 April 1994. Springer.

Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, second edition, 1988.

Kfoury, Assaf J., Harry Mairson, Franklyn Turbak, and Joe B. Wells. Relating typability and expressiveness in finite-rank intersection type systems. In *International Conference on Functional Programming (ICFP), Paris, France*, volume 34.9 of *ACM Sigplan Notices*, pages 90–101, N.Y., September 27–29 1999. ACM Press.

Kfoury, Assaf J. and Jerzy Tiuryn. Type reconstruction in finite-rank fragments of the polymorphic λ-calculus. In *Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 2–11, Philadelphia, PA, June 1990. Full version in *Information and Computation*, 98(2), 228–257, 1992.

Kfoury, Assaf J., Jerzy Tiuryn, and Pawel Urzyczyn. ML typability is DEXPTIME-complete. In *Proc. 15th Colloq. on Trees in Algebra and Programming*, pages 206–220. Springer LNCS 431, 1990.

Kfoury, Assaf J., Jerzy Tiuryn, and Pawel Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, April 1993a.

Kfoury, Assaf J., Jerzy Tiuryn, and Pawel Urzyczyn. The undecidability of the semi-unification problem. *Information and Computation*, 102(1):83–101, January 1993b. Summary in *STOC 1990*.

Kfoury, Assaf J., Jerzy Tiuryn, and Pawel Urzyczyn. An analysis of ML typability. *Journal of the ACM*, 41(2):368–398, March 1994.

Kfoury, Assaf J. and Joe B. Wells. Principality and decidable type inference for finite-rank intersection types. In *ACM Symposium on Principles of Programming Languages (POPL), San Antonio, Texas*, pages 161–174, New York, NY, January 1999. ACM.

Kiczales, Gregor, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.

Kirchner, Claude and Jean-Pierre Jouannaud. Solving equations in abstract algebras: a rule-based survey of unification. Research Report 561, Université de Paris Sud, Orsay, France, April 1990.

Klop, Jan W. *Combinatory Reduction Systems*. Mathematical Centre Tracts 127. Mathematisch Centrum, Amsterdam, 1980.

Kobayashi, Naoki, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, 1996. Full version in *ACM Transactions on Programming Languages and Systems*, 21(5), pp. 914-947, September 1999.

Kozen, Dexter, Jens Palsberg, and Michael I. Schwartzbach. Efficient recursive subtyping. In *ACM Symposium on Principles of Programming Languages (POPL), Charleston, South Carolina*, pages 419-428, 1993.

Laan, Twan Dismas Laurens. *The Evolution of Type Theory in Logic and Mathematics*. PhD thesis, Techn. Univ. Eindhoven, 1997.

Landin, Peter J. The mechanical evaluation of expressions. *Computer Journal*, 6: 308-320, January 1964.

Landin, Peter J. A correspondence between ALGOL 60 and Church's lambda-notation: Parts I and II. *Communications of the ACM*, 8(2,3):89-101, 158-165, February and March 1965.

Landin, Peter J. The next 700 programming languages. *Communications of the ACM*, 9(3):157-166, March 1966.

Lassez, Jean-Louis and Gordin Plotkin, editors. *Computational Logic, Essays in Honor of Alan Robinson*. MIT Press, 1991.

Läufer, Konstantin. *Polymorphic Type Inference and Abstract Data Types*. PhD thesis, New York University, 1992.

Läufer, Konstantin and Martin Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1411-1430, September 1994. Summary in *Phoenix Seminar and Workshop on Declarative Programming*, Nov. 1991.

League, Christopher, Zhong Shao, and Valery Trifonov. Representing Java classes in a typed intermediate language. In *International Conference on Functional Programming (ICFP), Paris, France*, September 1999.

League, Christopher, Valery Trifonov, and Zhong Shao. Type-preserving compilation of Featherweight Java. In *Foundations of Object-Oriented Languages (FOOL8)*, London, January 2001.

Lee, Oukseh and Kwangkeun Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20 (4):707-723, July 1998.

Leivant, Daniel. Polymorphic type inference. In *Proceedings of the 10th Annual ACM Symposium on Principles of Programming Languages*. ACM, 1983.

Lemmon, E. John, Carew A. Meredith, David Meredith, Arthur N. Prior, and Ivo Thomas. Calculi of pure strict implication, 1957. Mimeographed version, 1957; published in *Philosophical Logic*, ed. Davis, Hockney, and Wilson, D. Reidel Co., Netherlands, 1969, pp. 215-250.

Leroy, Xavier. Manifest types, modules and separate compilation. In *ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, pages 109-122, Portland, OR, January 1994.

Leroy, Xavier. The Objective Caml system: Documentation and user's manual, 2000. With Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. Available from `http://caml.inria.fr`.

Leroy, Xavier and Michel Mauny. Dynamics in ML. In John Hughes, editor, *ACM Symposium on Functional Programming Languages and Computer Architecture (FPCA) 1991*, volume 523 of *Lecture Notes in Computer Science*, pages 406-426. Springer-Verlag, *1991*.

Leroy, Xavier and François Pessaux. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems*, 22(2):340–377, March 2000. Summary in *ACM Symposium on Principles of Programming Languages (POPL), San Antonio, Texas*, 1999.

Leroy, Xavier and François Rouaix. Security properties of typed applets. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, pages 391–403, January 1998.

Leroy, Xavier and Pierre Weis. Polymorphic type inference and assignment. In *ACM Symposium on Principles of Programming Languages (POPL), Orlando, Florida*, pages 291–302, 1991.

Lescanne, Pierre and Jocelyn Rouyer-Degli. Explicit substitutions with de Bruijn's levels. In J. Hsiang, editor, *Proceedings of the 6th Conference on Rewriting Techniques and Applications (RTA), Kaiserslautern (Germany)*, volume 914, pages 294–308, 1995.

Levin, Michael Y. and Benjamin C. Pierce. Tinkertype: A language for playing with formal systems. *Journal of Functional Programming*, 2001. To appear. A preliminary version appeared as an invited talk at the *Logical Frameworks and Metalanguages Workshop (LFM)*, June 2000.

Lillibridge, Mark. *Translucent Sums: A Foundation for Higher-Order Module Systems.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1997.

Liskov, Barbara, Russell Atkinson, Toby Bloom, Elliott Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual.* Springer-Verlag, 1981.

Liskov, Barbara, Alan Snyder, Russell Atkinson, and J. Craig Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977. Also in S. Zdonik and D. Maier, eds., *Readings in Object-Oriented Database Systems.*

Luo, Zhaohui. *Computation and Reasoning: A Type Theory for Computer Science.* Number 11 in International Series of Monographs on Computer Science. Oxford University Press, 1994.

Luo, Zhaohui and Robert Pollack. The LEGO proof development system: A user's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, May 1992.

Ma, QingMing. Parametricity as subtyping. In *ACM Symposium on Principles of Programming Languages (POPL), Albuquerque, New Mexico*, January 1992.

Mackie, Ian. Lilac: A functional programming language based on linear logic. *Journal of Functional Programming*, 4(4):395–433, October 1994.

MacQueen, David. Using dependent types to express modular structure. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, pages 277–286, January 1986.

MacQueen, David, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71:95–130, 1986.

MacQueen, David B. Using dependent types to express modular structure. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, 1986.

Magnusson, Lena and Bengt Nordström. The ALF proof editor and its proof engine. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, pages 213–237. Springer-Verlag LNCS 806, 1994.

Mairson, Harry G. Deciding ML typability is complete for deterministic exponential time. In *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*, pages 382–401. ACM Press, New York, 1990.

Martin-Löf, Per. An intuitionistic theory of types: predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium, '73*, pages 73–118. North-Holland, Amsterdam, 1973.

Martin-Löf, Per. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science, VI*. North Holland, Amsterdam, 1982.

Martin-Löf, Per. *Intuitionistic Type Theory*. Bibliopolis, 1984.

Martini, Simone. Bounded quantifiers have interval models. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 174–183, Snowbird, Utah, July 1988. ACM.

McCarthy, John. History of LISP. In R. L. Wexelblatt, editor, *History of Programming Languages*, pages 173–197. Academic Press, New York, 1981.

McCarthy, John, S. R. Russell, D. Edwards, et al. *LISP Programmer's Manual*. Massachusetts Institute of Technology, A.I. Lab., Cambridge, Massachusetts, November 1959. Handwritten Draft + Machine Typed.

McKinna, James and Robert Pollack. Pure Type Sytems formalized. In M. Bezem and J. F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 289–305. Springer-Verlag LNCS 664, March 1993.

Meertens, Lambert. Incremental polymorphic type checking in B. In *ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas*, 1983.

Milner, Robin. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.

Milner, Robin. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

Milner, Robin. *Communication and Concurrency*. Prentice Hall, 1989.

Milner, Robin. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, October 1991. Appeared in *Proceedings of the International Summer School on Logic and Algebra of Specification*, Marktoberdorf, August 1991. Reprinted in *Logic and Algebra of Specification*, ed. F. L. Bauer, W. Brauer, and H. Schwichtenberg, Springer-Verlag, 1993.

Milner, Robin. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.

Milner, Robin, Joachim Parrow, and David Walker. A calculus of mobile processes (Parts I and II). *Information and Computation*, 100:1–77, 1992.

Milner, Robin and Mads Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87:209–220, 1991a.

Milner, Robin and Mads Tofte. *Commentary on Standard ML*. MIT Press, Cambridge, Massachusetts, 1991b.

Milner, Robin, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

Milner, Robin, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

Mitchell, John C. Coercion and type inference (summary). In *ACM Symposium on*

Principles of Programming Languages (POPL), Salt Lake City, Utah, pages 175-185, January 1984a.

Mitchell, John C. Type inference and type containment. In *Proc. Int. Symp. on Semantics of Data Types, Sophia-Antipolis (France)*, pages 257-278, Berlin, June 1984b. Springer LNCS 173. Full version in *Information and Computation*, vol. 76, no. 2/3, 1988, pp. 211-249. Reprinted in *Logical Foundations of Functional Programming*, ed. G. Huet, Addison-Wesley (1990) 153-194.

Mitchell, John C. Representation independence and data abstraction (preliminary version). In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, pages 263-276, 1986.

Mitchell, John C. Toward a typed foundation for method specialization and inheritance. In *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*, pages 109-124, January 1990a. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

Mitchell, John C. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, pages 365-458. North-Holland, Amsterdam, 1990b.

Mitchell, John C. *Foundations for Programming Languages.* MIT Press, Cambridge, Massachusetts, 1996.

Mitchell, John C. and Robert Harper. The essence of ML. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, January 1988. Full version in *ACM Transactions on Programming Languages and Systems*, vol. 15, no. 2, 1993, pp. 211-252, under the title "On the type structure of Standard ML".

Mitchell, John C. and Albert R. Meyer. Second-order logical relations (extended abstract). In Rohit Parikh, editor, *Logic of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 225-236, Berlin, 1985. Springer-Verlag.

Mitchell, John C. and Gordon D. Plotkin. Abstract types have existential types. *ACM Trans. on Programming Languages and Systems*, 10(3):470-502, 1988. Summary in *ACM Symposium on Principles of Programming Languages (POPL), New Orleans, Louisiana*, 1985.

Morris, James H. Lambda calculus models of programming languages. Technical Report MIT-LCS//MIT/LCS/TR-57, Massachusetts Institute of Technology, Laboratory for Computer Science, December 1968.

Morrisett, Greg, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 66-77, La Jolla, California, June 25-28, 1995. ACM SIGPLAN/SIGARCH and IFIP WG2.8, ACM Press.

Morrisett, Greg, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, pages 85-97, January 1998.

Mugridge, Warwick B., John Hamer, and John G. Hosking. Multi-methods in a statically-typed programming language. In Pierre America, editor, *ECOOP '91: European Conference on Object-Oriented Programming*, volume 512 of *Lecture Notes in Computer Science*, pages 307-324. Springer-Verlag, 1991.

Mycroft, Alan. Dynamic types in ML. Manuscript, 1983.

Mycroft, Alan. Polymorphic type schemes and recursive definitions. In M. Paul and B. Robinet, editors, *Proceedings of the International Symposium on Programming*, volume 167 of *LNCS*, pages 217–228, Toulouse, France, April 1984. Springer.

Myers, Andrew C., Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *ACM Symposium on Principles of Programming Languages (POPL), Paris, France*, pages 132–145, January 1997.

Nadathur, Gopalan and Dale Miller. An overview of λProlog. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 1*, pages 810–827, MIT Press, Cambridge, Massachusetts, August 1988.

Naur, Peter et al. Revised report on the algorithmic language Algol 60. *Communications of the ACM*, 6:1–17, January 1963.

Necula, George C. Proof-carrying code. In *ACM Symposium on Principles of Programming Languages (POPL), Paris, France*, pages 106–119, 15–17 January 1997.

Necula, George C. and Peter Lee. Safe kernel extensions without run-time checking. In *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996, Seattle, WA*, pages 229–243, Berkeley, CA, USA, October 1996. USENIX press.

Necula, George C. and Peter Lee. Safe, untrusted agents using proof-carrying code. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 61–91. Springer-Verlag, 1998.

Nelson, Greg, editor. *Systems Programming with Modula-3*. Prentice-Hall, 1991.

Nipkow, Tobias and David von Oheimb. $Java_{light}$ is type-safe — definitely. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, pages 161–170, January 1998.

O'Callahan, Robert and Daniel Jackson. Lackwit: A program understanding tool based on type inference. In *Proceedings of the 1997 International Conference on Software Engineering*, pages 338–348. ACM Press, 1997.

Odersky, Martin. Functional nets. In *Proc. European Symposium on Programming (ESOP)*, pages 1–25. Springer-Verlag, 2000. Lecture Notes in Computer Science 1782.

Odersky, Martin and Konstantin Läufer. Putting type annotations to work. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, pages 54–67, St. Petersburg, Florida, January 21–24, 1996. ACM Press.

Odersky, Martin, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999. Summary in *Workshop on Foundations of Object-Oriented Languages (FOOL), informal proceedings*, 1997.

Odersky, Martin and Philip Wadler. Pizza into Java: Translating theory into practice. In *ACM Symposium on Principles of Programming Languages (POPL), Paris, France*, pages 146–159, January 1997.

Odersky, Martin and Christoph Zenger. Nested types. In *Workshop on Foundations of Object-Oriented Languages (FOOL 8)*, January 2001.

Odersky, Martin, Christoph Zenger, and Matthias Zenger. Colored local type inference. *ACM SIGPLAN Notices*, 36(3):41–53, March 2001.

O'Hearn, Peter W., Makoto Takeyama, A. John Power, and Robert D. Tennent. Syntactic control of interference revisited. In *MFPS XI, conference on Mathematical Founda-*

tions of Program Semantics, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, March 1995.

O'Toole, James W. and David K. Gifford. Type reconstruction with first-class polymorphic values. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Portland, Oregon*, pages 207–217, June 1989.

Palsberg, Jens and Christina Pavlopoulou. From polyvariant flow information to intersection and union types. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, pages 197–208, 1998.

Palsberg, Jens and Michael I. Schwartzbach. *Object-Oriented Type Systems*. Wiley, 1994.

Park, David. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, Berlin, 1981.

Paulin-Mohring, Christine. Extracting F_ω's programs from proofs in the calculus of constructions. In *ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas*, pages 89–104, January 1989.

Paulson, Laurence C. *ML for the Working Programmer*. Cambridge University Press, New York, NY, second edition, 1996.

Perry, Nigel. *The Implementation of Practical Functional Programming Languages*. PhD thesis, Imperial College, 1990.

Peyton Jones, Simon L. and David R. Lester. *Implementing Functional Languages*. Prentice Hall, 1992.

Pfenning, Frank. Partial polymorphic type inference and higher-order unification. In *ACM Symposium on Lisp and Functional Programming (LFP), Snowbird, Utah*, pages 153–163, July 1988. Also available as Ergo Report 88–048, School of Computer Science, Carnegie Mellon University, Pittsburgh.

Pfenning, Frank. Elf: A language for logic definition and verified meta-programming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322, Pacific Grove, California, June 1989. IEEE Computer Society Press.

Pfenning, Frank. On the undecidability of partial polymorphic type reconstruction. *Fundamenta Informaticae*, 19(1,2):185–199, 1993a. Preliminary version available as Technical Report CMU-CS-92-105, School of Computer Science, Carnegie Mellon University, January 1992.

Pfenning, Frank. Refinement types for logical frameworks. In Herman Geuvers, editor, *Informal Proceedings of the Workshop on Types for Proofs and Programs*, pages 285–299, Nijmegen, The Netherlands, May 1993b.

Pfenning, Frank. Elf: A meta-language for deductive systems. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, pages 811–815, Nancy, France, June 1994. Springer-Verlag LNAI 814.

Pfenning, Frank. The practice of logical frameworks. In Hélène Kirchner, editor, *Proceedings of the Colloquium on Trees in Algebra and Programming*, pages 119–134, Linköping, Sweden, April 1996. Springer-Verlag LNCS 1059. Invited talk.

Pfenning, Frank. Logical frameworks. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier, 1999.

Pfenning, Frank. *Computation and Deduction*. Cambridge University Press, 2001.

Pfenning, Frank and Peter Lee. Metacircularity in the polymorphic λ-calculus. *The-*

oretical Computer Science, 89(1):137-159, 21 October 1991. Summary in *TAP-SOFT '89, Proceedings of the International Joint Conference on Theory and Practice in Software Development, Barcelona, Spain*, pages 345-359, Springer-Verlag LNCS 352, March 1989.

Pierce, Benjamin C. *Basic Category Theory for Computer Scientists.* MIT Press, 1991a.

Pierce, Benjamin C. *Programming with Intersection Types and Bounded Polymorphism.* PhD thesis, Carnegie Mellon University, December 1991b. Available as School of Computer Science technical report CMU-CS-91-205.

Pierce, Benjamin C. Bounded quantification is undecidable. *Information and Computation*, 112(1):131-165, July 1994. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994). Summary in *ACM Symposium on Principles of Programming Languages (POPL), Albuquerque, New Mexico.*

Pierce, Benjamin C. Even simpler type-theoretic foundations for OOP. Manuscript (circulated electronically), March 1996.

Pierce, Benjamin C. Bounded quantification with bottom. Technical Report 492, Computer Science Department, Indiana University, 1997a.

Pierce, Benjamin C. Intersection types and bounded polymorphism. *Mathematical Structures in Computer Science*, 7(2):129-193, April 1997b. Summary in *Typed Lambda Calculi and Applications*, March 1993, pp. 346-360.

Pierce, Benjamin C. and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Logic in Computer Science*, 1993. Full version in *Mathematical Structures in Computer Science* , Vol. 6, No. 5, 1996.

Pierce, Benjamin C. and Martin Steffen. Higher-order subtyping. In *IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET)*, 1994. Full version in *Theoretical Computer Science*, vol. 176, no. 1-2, pp. 235-282, 1997 (corrigendum in TCS vol. 184 (1997), p. 247).

Pierce, Benjamin C. and David N. Turner. Statically typed friendly functions via partially abstract types. Technical Report ECS-LFCS-93-256, University of Edinburgh, LFCS, April 1993. Also available as INRIA-Rocquencourt Rapport de Recherche No. 1899.

Pierce, Benjamin C. and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207-247, April 1994. Summary in *ACM Symposium on Principles of Programming Languages (POPL), Charleston, South Carolina*, 1993.

Pierce, Benjamin C. and David N. Turner. Local type argument synthesis with bounded quantification. Technical Report 495, Computer Science Department, Indiana University, January 1997.

Pierce, Benjamin C. and David N. Turner. Local type inference. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, 1998. Full version in *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1), January 2000, pp. 1-44.

Pierce, Benjamin C. and David N. Turner. Pict: A programming language based on the pi-calculus. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 455-494. MIT Press, 2000.

Pitts, Andrew M. Polymorphism is set theoretic, constructively. In Pitt, Poigné, and

Rydeheard, editors, *Category Theory and Computer Science, Edinburgh*, pages 12-39. Springer-Verlag, 1987. LNCS volume 283.

Pitts, Andrew M. Non-trivial power types can't be subtypes of polymorphic types. In *Fourth Annual Symposium on Logic in Computer Science, Pacific Grove, California*, pages 6-13. IEEE, June 1989.

Pitts, Andrew M. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10:321-359, 2000.

Plasmeijer, Marinus J. CLEAN: a programming environment based on term graph rewriting. *Theoretical Computer Science*, 194(1-2), March 1998.

Plotkin, Gordon. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1:125-159, 1975.

Plotkin, Gordon and Martín Abadi. A logic for parametric polymorphism. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications (TLCA), Utrecht, The Netherlands*, number 664 in Lecture Notes in Computer Science, pages 361-375. Springer-Verlag, March 1993.

Plotkin, Gordon, Martín Abadi, and Luca Cardelli. Subtyping and parametricity. In *Proceedings of the Ninth IEEE Symposium on Logic in Computer Science*, pages 310-319, 1994.

Plotkin, Gordon D. Lambda-definability and logical relations. Memorandum SAI-RM-4, University of Edinburgh, Edinburgh, Scotland, October 1973.

Plotkin, Gordon D. LCF considered as a programming language. *Theoretical Computer Science*, 5:223-255, 1977.

Plotkin, Gordon D. Lambda-definability in the full type hierarchy. In Jonathan P. Seldin and J. Roger Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 363-373. Academic Press, London, 1980.

Plotkin, Gordon D. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981.

Poll, Erik. Width-subtyping and polymorphic record update. Manuscript, June 1996.

Pollack, Robert. Implicit syntax. Informal Proceedings of First Workshop on Logical Frameworks, Antibes, May 1990.

Pollack, Robert. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1994.

Pottier, François. Simplifying subtyping constraints. In *International Conference on Functional Programming (ICFP), Amsterdam, The Netherlands*, 1997.

Pottinger, Garrell. A type assignment for the strongly normalizable λ-terms. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 561-577. Academic Press, New York, 1980.

Quine, Willard V. *Quiddities: An Intermittently Philosophical Dictionary.* Harvard University Press, Cambridge, MA, 1987.

Ramsey, Frank P. The foundations of mathematics. *Proceedings of the London Mathematical Society, Series 2*, 25(5):338-384, 1925. Reprinted in (Braithwaite, 1931).

Ranta, Aarne. *Type-Theoretical Grammar.* Clarendon Press, Oxford, 1995.

Reade, Chris. *Elements of Functional Programming.* International Computer Science

Series. Addison-Wesley, Wokingham, England, 1989.

Reddy, Uday S. Objects as closures: Abstract semantics of object oriented languages. In *ACM Symposium on Lisp and Functional Programming (LFP), Snowbird, Utah*, pages 289–297, Snowbird, Utah, July 1988.

Relax. Document Description and Processing Languages — Regular Language Description for XML (RELAX) — Part 1: RELAX Core. Technical Report DTR 22250-1, ISO/IEC, October 2000.

Rémy, Didier. Typechecking records and variants in a natural extension of ML. In *ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas*, pages 242–249, January 1989. Long version in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

Rémy, Didier. *Algèbres Touffues. Application au Typage Polymorphe des Objets Enregistrements dans les Langages Fonctionnels.* PhD thesis, Université Paris VII, 1990.

Rémy, Didier. Extending ML type system with a sorted equational theory. Research Report 1766, Institut National de Recherche en Informatique et Automatisme, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France, 1992a.

Rémy, Didier. Projective ML. In *ACM Symposium on Lisp and Functional Programming (LFP)*, pages 66–75, 1992b.

Rémy, Didier. Typing record concatenation for free. In *ACM Symposium on Principles of Programming Languages (POPL), Albuquerque, New Mexico*, January 1992. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

Rémy, Didier. Programming objects with ML-ART: An extension to ML with abstract and record types. In Masami Hagiya and John C. Mitchell, editors, *International Symposium on Theoretical Aspects of Computer Software (TACS)*, pages 321–346, Sendai, Japan, April 1994. Springer-Verlag.

Rémy, Didier. *Des enregistrements aux objets.* Mémoire d'habilitation à diriger des recherches, Université de Paris 7, 1998. In English, except for introductory chapter; includes (Rémy, 1989) and (Rémy, 1992b).

Rémy, Didier and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Object Systems*, 4(1):27–50, 1998. Summary in *ACM Symposium on Principles of Programming Languages (POPL), Paris, France*, 1997.

Reynolds, John. Three approaches to type structure. In *Mathematical Foundations of Software Development*. Springer-Verlag, 1985. Lecture Notes in Computer Science No. 185.

Reynolds, John C. Towards a theory of type structure. In *Proc. Colloque sur la Programmation*, pages 408–425, New York, 1974. Springer-Verlag LNCS 19.

Reynolds, John C. User-defined types and procedural data structures as complementary approaches to data abstraction. In Stephen A. Schuman, editor, *New Directions in Algorithmic Languages 1975*, pages 157–168, Rocquencourt, France, 1975. IFIP Working Group 2.1 on Algol, INRIA. Reprinted in (Gries, 1978, pages 309–317) and (Gunter and Mitchell, 1994, pages 13–23).

Reynolds, John C. Syntactic control of interference. In *ACM Symposium on Principles of Programming Languages (POPL), Tucson, Arizona*, pages 39–46, 1978. Reprinted in O'Hearn and Tennent, *ALGOL-like Languages*, vol. 1, pages 273–286, Birkhäuser, 1997.

Reynolds, John C. Using category theory to design implicit conversions and generic operators. In N. D. Jones, editor, *Proceedings of the Aarhus Workshop on Semantics-Directed Compiler Generation*, number 94 in Lecture Notes in Computer Science. Springer-Verlag, January 1980. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

Reynolds, John C. *The Craft of Programming.* Prentice-Hall International, London, 1981.

Reynolds, John C. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513-523, Amsterdam, 1983. Elsevier Science Publishers B. V. (North-Holland).

Reynolds, John C. Polymorphism is not set-theoretic. In G. Kahn, D. B. MacQueen, and G. D. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 145-156, Berlin, 1984. Springer-Verlag.

Reynolds, John C. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, June 1988. Reprinted in O'Hearn and Tennent, *ALGOL-like Languages*, vol. 1, pages 173-233, Birkhäuser, 1997.

Reynolds, John C. Syntactic control of interference, part 2. Report CMU-CS-89-130, Carnegie Mellon University, April 1989.

Reynolds, John C. Introduction to part II, polymorphic lambda calculus. In Gérard Huet, editor, *Logical Foundations of Functional Programming*, pages 77-86. Addison-Wesley, Reading, Massachusetts, 1990.

Reynolds, John C. The coherence of languages with intersection types. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software (Sendai, Japan)*, number 526 in Lecture Notes in Computer Science, pages 675-700. Springer-Verlag, September 1991.

Reynolds, John C. Normalization and functor categories. In Olivier Danvy and Peter Dybjer, editors, *Preliminary Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation, NBE '98*, (Chalmers, Sweden, May 8-9, 1998), number NS-98-1 in BRICS Note Series, Department of Computer Science, University of Aarhus, May 1998a.

Reynolds, John C. *Theories of Programming Languages.* Cambridge University Press, 1998b.

Reynolds, John C. and Gordon Plotkin. On functors expressible in the polymorphic typed lambda calculus. *Information and Computation*, 105(1):1-29, 1993. Summary in (Huet, 1990).

Robinson, Edmund and Robert Tennent. Bounded quantification and record-update problems. Message to Types electronic mail list, October 1988.

Robinson, J. Alan. Computational logic: The unification computation. *Machine Intelligence*, 6:63-72, 1971.

Russell, Bertrand. Letter to Frege, 1902. Reprinted (in English) in J. van Heijenort, editor, *From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931*; Harvard University Press, Cambridge, MA, 1967; pages 124-125.

Schaffert, Justin Craig. A formal definition of CLU. Master's thesis, MIT, January 1978. MIT/LCS/TR-193.

Scheifler, Robert William. A denotational semantics of CLU. Master's thesis, MIT, May 1978. MIT/LCS/TR-201.

Schmidt, David A. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, 1986.

Schmidt, David A. *The Structure of Typed Programming Languages.* MIT Press, 1994.

Schönfinkel, Moses. Über die Bausteine der mathematischen Logik. *Mathematische Annalen,* 92:305-316, 1924. Translated into English and republished as "On the building blocks of mathematical logic" in (van Heijenoort, 1967, pp. 355-366).

Scott, Michael L. *Programming Language Pragmatics.* Morgan Kaufmann, 1999.

Severi, Paula and Erik Poll. Pure type systems with definitions. In *Proceedings of Logical Foundations of Computer Science (LFCS),* pages 316-328. Springer-Verlag, 1994. LNCS volume 813.

Shalit, Andrew. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language.* Addison-Wesley, Reading, Mass., 1997.

Shields, Mark. *Static Types for Dynamic Documents.* PhD thesis, Department of Computer Science, Oregon Graduate Institute, February 2001.

Simmons, Harold. *Derivation and Computation : Taking the Curry-Howard Correspondence Seriously.* Number 51 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2000.

Smith, Frederick, David Walker, and Greg Morrisett. Alias types. In Gert Smolka, editor, *Ninth European Symposium on Programming,* volume 1782 of *Lecture Notes in Computer Science,* pages 366-381. Springer-Verlag, April 2000.

Smith, Jan, Bengt Nordström, and Kent Petersson. *Programming in Martin-Löf's Type Theory. An Introduction.* Oxford University Press, 1990.

Solomon, Marvin. Type definitions with parameters. In *ACM Symposium on Principles of Programming Languages (POPL), Tucson, Arizona,* pages 31-38, January 23-25, 1978.

Sommaruga, Giovanni. *History and Philosophy of Constructive Type Theory,* volume 290 of *Synthese Library.* Kluwer Academic Pub., 2000.

Somogyi, Zoltan, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming,* 29(1-3):17-64, October-November 1996.

Sørensen, Morten Heine and Paweł Urzyczyn. Lectures on the Curry-Howard isomorphism. Technical Report 98/14 (= TOPPS note D-368), DIKU, Copenhagen, 1998.

Statman, Richard. Completeness, invariance and λ-definability. *Journal of Symbolic Logic,* 47(1):17-26, 1982.

Statman, Richard. Equality between functionals, revisited. In *Harvey Friedman's Research on the Foundations of Mathematics,* pages 331-338. North-Holland, Amsterdam, 1985a.

Statman, Richard. Logical relations and the typed λ-calculus. *Information and Control,* 65(2-3):85-97, May-June 1985b.

Steffen, Martin. *Polarized Higher-Order Subtyping.* PhD thesis, Universität Erlangen-Nürnberg, 1998.

Stone, Christopher A. and Robert Harper. Deciding type equivalence in a language with singleton kinds. In *ACM Symposium on Principles of Programming Languages (POPL), Boston, Massachusetts,* pages 214-227, January 19-21, 2000.

Strachey, Christopher. Fundamental concepts in programming languages. Lecture Notes, International Summer School in Computer Programming, Copenhagen, August 1967. Reprinted in *Higher-Order and Symbolic Computation*, 13(1/2), pp. 1-49, 2000.

Stroustrup, Bjarne. *The C++ Programming Language.* Addison Wesley Longman, Reading, MA, third edition, 1997.

Studer, Thomas. Constructive foundations for featherweight java. In R. Kahle, P. Schroeder-Heister, and R. Stärk, editors, *Proof Theory in Computer Science.* Springer-Verlag, 2001. Lecture Notes in Computer Science, volume 2183.

Sumii, Eijiro and Benjamin C. Pierce. Logical relations for encryption. In *Computer Security Foundations Workshop*, June 2001.

Sussman, Gerald Jay and Guy Lewis Steele, Jr. Scheme: an interpreter for extended lambda calculus. MIT AI Memo 349, Massachusetts Institute of Technology, December 1975. Reprinted, with a foreword, in *Higher-Order and Symbolic Computation*, 11(4), pp. 405-439, 1998.

Syme, Don. Proving Java type soundness. Technical Report 427, Computer Laboratory, University of Cambridge, June 1997.

Tait, William W. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic*, 32(2):198-212, June 1967.

Tait, William W. A realizability interpretation of the theory of species. In R. Parikh, editor, *Logic Colloquium*, volume 453 of *Lecture Notes in Mathematics*, pages 240-251, Boston, 1975. Springer-Verlag.

Talpin, Jean-Pierre and Pierre Jouvelot. The type and effects discipline. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 162-173, 1992.

Tarditi, David, Greg Morrisett, Perry Cheng, Christopher Stone, Robert Harper, and Peter Lee. TIL : A type-directed optimizing compiler for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Philadephia, Pennsylvania*, pages 181-192, May 21-24 1996.

Tarski, Alfred. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285-309, 1955.

Tennent, Robert D. *Principles of Programming Languages.* Prentice-Hall, 1981.

Terlouw, J. Een nadere bewijstheoretische analyse van GSTTs. Manuscript, University of Nijmegen, Netherlands, 1989.

Thatte, Satish R. Quasi-static typing (preliminary report). In *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*, pages 367-381, 1990.

Thompson, Simon. *Type Theory and Functional Programming.* Addison Wesley, 1991.

Thompson, Simon. *Haskell: The Craft of Functional Programming.* Addison Wesley, 1999.

Tiuryn, Jerzy. Type inference problems: A survey. In B. Rovan, editor, *Mathematical Foundations of Computer Science 1990, Banskà Bystrica, Czechoslovakia*, volume 452 of *Lecture Notes in Computer Science*, pages 105-120. Springer-Verlag, New York, NY, 1990.

Tofte and Birkedal. A region inference algorithm. *ACMTOPLAS: ACM Transactions on Programming Languages and Systems*, 20, 1998.

Tofte, Mads. Type inference for polymorphic references. *Information and Computation*, 89(1), November 1990.

Tofte, Mads and Jean-Pierre Talpin. Implementing the call-by-value lambda-calculus using a stack of regions. In *ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, January 1994.

Tofte, Mads and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1 February 1997.

Trifonov, Valery and Scott Smith. Subtyping constrained types. In *Proceedings of the Third International Static Analysis Symposium*, volume 1145 of *LNCS*, pages 349–365. Springer-Verlag, September 1996.

Turner, David N., Philip Wadler, and Christian Mossin. Once upon a type. In *ACM Symposium on Functional Programming Languages and Computer Architecture (FPCA)*, San Diego, California, 1995.

Turner, Raymond. *Constructive Foundations for Functional Languages.* McGraw Hill, 1991.

Ullman, Jeffrey D. *Elements of ML Programming.* Prentice-Hall, ML97 edition, 1997.

Ungar, David and Randall B. Smith. Self: The power of simplicity. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 227–241, 1987.

U.S. Dept. of Defense. *Reference Manual for the Ada Programming Language.* GPO 008-000-00354-8, 1980.

van Benthem, Johan. *Language in Action: Categories, Lambdas, and Dynamic Logic.* MIT Press, 1995.

van Benthem, Johan F. A. K. and Alice Ter Meulen, editors. *Handbook of Logic and Language.* MIT Press, 1997.

van Heijenoort, Jan, editor. *From Frege to Gödel.* Harvard University Press, Cambridge, Massachusetts, 1967.

van Wijngaarden, Adriaan, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker. Revised report on the algorithmic language ALGOL 68. *Acta Informatica*, 5(1–3):1–236, 1975.

Vouillon, Jérôme. *Conception et réalisation d'une extension du langage ML avec des objets.* PhD thesis, Université Paris 7, October 2000.

Vouillon, Jérôme. Combining subsumption and binary methods: An object calculus with views. In *ACM Symposium on Principles of Programming Languages (POPL), London, England*, 2001.

Wadler, Philip. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM Press, September 1989. Imperial College, London.

Wadler, Philip. Linear types can change the world. In *TC 2 Working Conference on Programming Concepts and Methods (Preprint)*, pages 546–566, 1990.

Wadler, Philip. Is there a use for linear logic? In *Proceedings of ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 255–273, 1991.

Wadler, Philip. New languages, old logic. *Dr. Dobbs Journal*, December 2000.

Wadler, Philip. The Girard-Reynolds isomorphism. In Naoki Kobayashi and Benjamin Pierce, editors, *Theoretical Aspects of Computer Software (TACS), Sendai, Japan*, Lecture Notes in Computer Science. Springer-Verlag, 2001.

Wadler, Philip and Stephen Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In

ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas, pages 60–76, 1989.

Wadsworth, Christopher P. *Semantics and pragmatics of the lambda-calculus.* PhD thesis, Programming Research Group, Oxford University, 1971.

Wand, Mitchell. Finding the source of type errors. *13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL),* pages 38–43, 1986.

Wand, Mitchell. Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science,* Ithaca, NY, June 1987.

Wand, Mitchell. Corrigendum: Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science,* 1988.

Wand, Mitchell. Type inference for objects with instance variables and inheritance. Technical Report NU-CCS-89-2, College of Computer Science, Northeastern University, February 1989a. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

Wand, Mitchell. Type inference for record concatenation and multiple inheritance. In *Fourth Annual IEEE Symposium on Logic in Computer Science,* pages 92–97, Pacific Grove, CA, June 1989b.

Weis, Pierre, María-Virginia Aponte, Alain Laville, Michel Mauny, and Ascánder Suárez. The CAML reference manual, Version 2.6. Technical report, Projet Formel, INRIA-ENS, 1989.

Wells, Joe B. Typability and type checking in the second-order λ-calculus are equivalent and undecidable. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science (LICS),* pages 176–185, 1994.

Whitehead, Alfred North and Bertrand Russell. *Principia Mathematica.* Cambridge University Press, Cambridge, 1910. Three volumes (1910; 1912; 1913).

Wickline, Philip, Peter Lee, Frank Pfenning, and Rowan Davies. Modal types as staging specifications for run-time code generation. *ACM Computing Surveys,* 30(3es), September 1998. Article 8.

Wille, Christoph. *Presenting C#.* SAMS Publishing, 2000.

Winskel, Glynn. *The Formal Semantics of Programming Languages: An Introduction.* MIT Press, 1993.

Wirth, Niklaus. The programming language Pascal. *Acta Informatica,* 1(1):35–63, 1971.

Wright, Andrew K. Typing references by effect inference. In Bernd Krieg-Bruckner, editor, *ESOP '92, 4th European Symposium on Programming, Rennes, France,* volume 582 of *Lecture Notes in Computer Science,* pages 473–491. Springer-Verlag, New York, N.Y., 1992.

Wright, Andrew K. Simple imperative polymorphism. *Lisp and Symbolic Computation,* 8(4):343–355, 1995.

Wright, Andrew K. and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation,* 115(1):38–94, 15 November 1994.

Xi, Hongwei and Robert Harper. A dependently typed assembly language. In *International Conference on Functional Programming (ICFP), Firenze, Italy,* 2001.

Xi, Hongwei and Frank Pfenning. Eliminating array bound checking through depen-

dent types. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada*, pages 249–257, 1998.

Xi, Hongwei and Frank Pfenning. Dependent types in practical programming. In *ACM Symposium on Principles of Programming Languages (POPL), San Antonio, Texas*, ACM SIGPLAN Notices, pages 214–227, 1999.

XML 1998. Extensible markup language (XML™), February 1998. XML 1.0, W3C Recommendation, `http://www.w3.org/XML/`.

XS 2000. XML Schema Part 0: Primer, W3C Working Draft. `http://www.w3.org/TR/xmlschema-0/`, 2000.

Yelick, Kathy, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: a high-performance Java dialect. *Concurrency: Practice and Experience*, 10(11–13):825–836, September 1998. Special Issue: Java for High-performance Network Computing.

Zwanenburg, Jan. Pure type systems with subtyping. In J.-Y. Girard, editor, *Typed Lambda Calculus and Applications (TLCA)*, pages 381–396. Springer-Verlag, 1999. Lecture Notes in Computer Science, volume 1581.

书名=类型和程序设计语言
作者=（美）皮尔斯著　马世龙　眭跃飞等译
页数=422
SS号=11393905
出版日期=2005年05月
出版社=电子工业出版社
书名页数=1
版权页数=1
前言页数=1
目录页数=7
封面页数=2